N° d'ordre : 8754

THÈSE

DE L'UNIVERSITÉ PARIS–SUD

presentée en vue de l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ PARIS–SUD

Specialité : INFORMATIQUE

par

SYLVAIN GELLY

UNE CONTRIBUTION À L'APPRENTISSAGE PAR RENFORCEMENT ; APPLICATION AU COMPUTER-GO

Soutenue le 25 Septembre 2007 devant la commission d'examen :

M.	Jacques BLANC-TALON	DGA	Examinateur
M.	Olivier BOUSQUET	Google	Examinateur
M.	Nicolas BREDECHE	Université Paris-Sud	Co-Directeur de thèse
M.	Rémi MUNOS	INRIA	Rapporteur
Mme	Michèle SEBAG	CNRS	Directrice de thèse
M.	John SHAWE-TAYLOR	University College London	Examinateur
M.	Csaba SZEPESVARI	University of Alberta	Rapporteur

Ceci est un résumé en français du manuscrit de thèse. Le manuscrit complet (rédigé en anglais) se situe à la suite de ce résumé.

Following is a french summary of the thesis. The complete manuscrit, written in english, is found after this summary.

Résumé Français

Introduction

Le domaine de l'Apprentissage par Renforcement (AR) [SB98, BT96] se trouve à l'interface entre la théorie du contrôle, l'apprentissage supervisé et non-supervisé, l'optimisation et les sciences cognitives. Alors que l'AR s'attaque à des objectifs ayant des impacts économiques majeurs (par exemple le marketing [AVAS04], le contrôle de centrales électriques [SDG$^+$00], les bio-réacteurs [CVPL05], le contrôle d'hélicoptères [NKJS04], la finance [MS01], les jeux vidéos [SHJ01b], la robotique [KS04]), les difficultés théoriques et pratiques sont nombreuses.

Formellement, l'AR considère un *environnement* décrit à partir de *transitions* ; étant donné l'état courant du système étudié et l'action choisie par le contrôleur, la transition définit l'*état* suivant. L'environnement définit aussi la *récompense* perçue par le système lorsqu'il évolue à travers les états visités. Le but de l'AR est de trouver une *politique* ou contrôleur, associant à chaque état une action de telle sorte qu'elle optimise les récompenses reçues au court du temps. L'AR apprend une politique par exploration de l'espace d'état $\times$ actions, soit réellement, soit en simulation.

Ce manuscript présente les contributions de cette thèse dans le domaine de l'Apprentissage par Renforcement :

- Le châpitre 2 est consacré aux Réseaux Bayésiens (RBs), qui sont communément utilisés comme outils de représentation des environnements en AR. Notre contribution théorique est basée sur une nouvelle borne des nombres de couverture de l'espace des RBs, incluant l'*entropie structurelle* du réseau, en plus du classique nombre de paramètres. Les contributions algorithmiques concernent l'optimisation

paramétrique et non paramétrique du critère d'apprentissage basé sur cette borne, dont les mérites sont démontrés empiriquement sur des problèmes artificiels.

- Le châpitre 3 étudie la Programmation Dynamique Stochastique, en se concentrant sur les espaces continus d'états et d'actions. Les contributions concernent l'optimisation non-linéaire, la regression et l'échantillonnage au travers d'études expérimentales. Ces études sont toutes conduites dans notre plateforme OpenDP, un outil pour comparer les algorithmes sur différents problèmes.

- Le châpitre 4 s'intéresse à un problème discret en grande dimension, le jeu de Go, qui est communément considéré comme un des challenges majeurs à la fois en apprentissage automatique et en intelligence artificielle [BC01]. Un nouvel algorithme, inspiré du domaine de la théorie des jeux, appelé UCT [KS06], est adapté au contexte du Go. L'approche résultante, le programme MoGo, est le programme de Go le plus performant au moment de l'écriture de cette thèse, et ouvre de prometteuses perspectives de recherche.

La suite de ce résumé est organisé de la façon suivante. La première section présente brièvement des notions fondamentales du domaine de l'AR. La section suivante présente les contributions dans le domaine de l'apprentissage de modèle sous la forme de Réseaux Bayesiens. Puis se trouve un résumé des résultats correspondant à la programmation dynamique dans le continu. A la suite se trouvent les contributions dans le domaine applicatif du jeu de Go. Enfin la dernière section conclut et donne des perspectives à ce travail.

Notions Fondamentales

Markov Decision Process (MDP)

Les Processus de Décision de Markov (PDMs, *Markov Decision Process* en anglais) est le concept le plus courant pour modéliser les problèmes d'apprentissage par renforcement. Formellement, un PDM est défini comme un quadruplet $(\mathcal{S}, \mathcal{A}, p, r)$ où:

- $\mathcal{S}$ est l'espace d'état (peut être discret ou continu);

- $\mathcal{A}$ est l'espace d'action (peut être discret ou continu);
- p distribution de probabilité de la fonction de transition;
- r est la fonction de récompense.

Le temps est supposé discret dans la suite. p est définie pour tout pas de temps t (on note p_t), et son domaine de définition est $\mathcal{S} \times \mathcal{S} \times \mathcal{A}$. Plus précisement, pour un pas de temps t, un état $s \in \mathcal{S}$ et une action $a \in \mathcal{A}$, $p_t(.|s,a)$ est la distribution de probabilité de l'état suivant que l'on peut atteindre en prenant l'action a dans l'état s. En particulier,

$$\forall t,\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A}, \int_{s' \in \mathcal{S}} p_t(s'|s,a) = 1 ds'$$

La fonction de récompense r est définie pour tout pas de temps t (on note r_t), et son domaine de définition est $\mathcal{S} \times \mathcal{A}$. Plus précisément pour un pas de temps donné t, un état $s \in \mathcal{S}$ et une action $a \in \mathcal{A}$, $r_t(s,a)$ est la récompense immédiate obtenue en prenant l'action a dans l'état s. $r_t(s,a)$ peut être aléatoire.

Certains PDMs peuvent être *stationaires*, i.e. p et r ne dépendent pas de t. Plus précisement,

$$\exists p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R},\ \forall t\ p_t = p$$

et

$$\exists r : \mathcal{S} \times \mathcal{A} \to \mathbb{R},\ \forall t\ r_t = r$$

La propriété principale des PDMs (stationaires ou pas) et la *propriété de Markov*, qui dit que la distribution des états suivants et les récompenses ne dépendent que de l'état courant et de l'action et pas des états et actions passés. En notant un historique $h_t = (s_0, a_0, ..., s_{t-1}, a_{t-1})$ et H_t l'ensemble des historiques pour un pas de temps donné t, la propriété de Markov s'écrit:

$$\forall t,\ \forall h_t \in H_t,\ \forall s_t, s_{t+1} \in \mathcal{S},\ \forall a_t \in \mathcal{A},\ P(s_{t+1}|h_t, s_t, a_t) = P(s_{t+1}|s_t, a_t) = p_t(s_{t+1}|s_t, a_t)$$

Politique

Une *politique* modélise les décisions prises par l'agent dans l'environnement. Une politique est une fonction dont le domaine de définition est $\mathcal{S}$, souvent notée π, et pour $s \in \mathcal{S}$, $\forall t\ \pi_t(s)$ est une variable aléatoire sur $\mathcal{A}$.

Une politique est *stationnaire* quand elle ne dépend pas du pas de temps t, i.e. $\exists \pi,\ \forall t,\ \forall s \in \mathcal{S}\ \pi_t(s) = \pi(s)$.

Critères

Pour résoudre un PDM, c'est à dire construire une "bonne" politique pour ce problème, se pose la question du critère à maximiser. Les critères principaux sont une somme des récompenses instantannées, éventuellement actualisées (c'est à dire quand les récompenses futures sont exponentiellement diminuées par γ^t avec t la distance au moment futur et $\gamma < 1$). Voici les deux principaux critères utilisés dans la suite:

- Horizon fini T: $E[\sum_{t=0}^{T-1} r_t]$
- γ-actualisé: $E[\sum_{t=0}^{\infty} \gamma^t r_t]$

Fonction de valeur

La fonction de valeur d'une politique π donnée en un état s est la valeur du critère choisi si l'agent part de cet état s et suit la politique π jusqu'à la fin de l'épisode (si horizon fini ou état absorbant), ou la limite en fonction du temps (si horizon infini).

Programmation Dynamique Stochastique (horizon fini)

Un algorithme classique à la base de nombreux algorithmes d'apprentissage par renforcement est la programmation dynamique [Bel57]. Nous le présentons ici par son importance et à cause de son utilisation dans la suite de la thèse. Pour simplifier, on se place dans le cadre horizon fini.

Le modèle de l'environnement est ici supposé connu (i.e. on peut simuler chaque transition en n'importe quel état, sachant l'action). Le principe d'optimalité de Bellman dit

que l'accès à la fonction de valeur de la politique optimale pour le pas de temps $t+1$ est suffisant pour calculer la fonction de valeur de la politique optimale pour le pas de temps t. Comme la fonction de valeur de la politique optimale pour le dernier pas de temps est triviale à calculer, par récurrence à l'envers dans le temps, on peut calculer la fonction de valeur de la politique optimale pour tous les pas de temps. A partir de cette fonction de valeur, la politique optimale est simplement celle qui est gloutonne sur cette fonction de valeur. L'algorithme 1 illustre le principe dans le cas simple où les espaces d'états et d'actions sont finis.

Algorithm 1 Programmation dynamique en espace d'états et d'actions finis

Entrée: un PDM, avec $\mathcal{S}$ et $\mathcal{A}$ finis, horizon fini T.
Initialiser $\forall s \in \mathcal{S},\ V_T(s) = 0$
for $t = T-1$ **descendant vers** 0 **do**
  for tout $s \in \mathcal{S}$ **do**
    $V_t(s) = \sup_{a\in\mathcal{A}} \left(r_t(s,a) + \int_{s'\in\mathcal{S}} p_t(s'|s,a)V_{t+1}(s')ds'\right)$
  end for
end for
Sortie: $\forall t,\ V_t = V_t^*$.
Sortie: $\forall t, \forall s\ \pi_t(s) = \arg\max_{a\in\mathcal{A}} \left(r_t(s,a) + \int_{s'\in\mathcal{S}} p_t(s'|s,a)V_{t+1}(s')ds'\right) = \pi_t^*(s)$.
Complexité: $T \times |\mathcal{S}|^2 \times |\mathcal{A}|$.

Si $\mathcal{S}$ est continu ou trop grand, $V_t(s)$ ne peut plus être calculé pour tout $s \in \mathcal{S}$, et le calcul de l'intégrale n'est plus trivial. De plus si $\mathcal{A}$ est continu ou trop grand, le calcul de $\sup_{a\in\mathcal{A}}$ devient lui aussi difficile. Cela fait l'objet du Chapitre 3.

Modèles et Réseaux Bayesiens

Introduction

La représentation de l'environnement, i.e. le modèle de transition et des récompenses, joue un rôle clé dans le succès de nombreuses méthodes de résolution du problème d'AR. Nous nous intéressons plus particulièrement ici au formalisme des Réseaux Bayesiens (RBs) [KP00] qui s'est imposé comme un formalisme populaire pour représenter de façon compacte (factorisée) des distributions de probabilités en grande dimension. Le cas particulier

des Réseaux Bayesiens Dynamiques [Mur02] peut alors servir à représenter des PDMs.

Les RBs sont issus d'un mariage entre la théorie des graphes et celle des probabilités. Un RB est un graphe dirigé sans circuit, dans lequel chaque noeud représente une variable aléatoire, et les dépendances conditionnelles sont représentées par les arcs. Plus précisément, si $A_1,...,A_n$ sont les variables aléatoires (on les confond avec les noeuds du graphe), ordonnées dans un ordre topologique du graphe, et $Pa(A_i)$ l'ensemble des parents de A_i dans le graphe, alors on peut écrire:

$$P(A_1,...,A_n) = \prod_{i=1}^{n} P(A_i|Pa(A_i))$$

Les contributions principales dans ce domaine sont les suivantes:

- La définition d'un nouveau critère d'apprentissage, distinguant l'apprentissage paramétrique et non paramétrique (apprendre les paramètres ou la structure du RB).
- La preuve théorique et expérimentale des propriétés de robustesse (par rapport à la structure) de ce critère.
- La proposition de plusieurs algorithmes pour assurer la faisabilité de l'optimisation.
- Des bornes non asymptotiques sur l'erreur d'apprentissage.

Problématique

Apprendre un RBs peut être fait dans plusieurs buts: (i) évaluer qualitativement des probabilités, où la question "cet événement apparait-il avec probabilité 10^{-30} ou 10^{-5} ?" prend tout son sens; (ii) utiliser le RB pour calculer des espérances (gains ou pertes d'une stratégie $f(X)$ par exemple).

Dans le premier cas, l'évaluation d'un risque, la mesure des erreurs d'apprentissage grâce à des logarithmes est naturelle, ce qui amène aux approches de maximum de vraisemblance.

Dans le second cas, on veut estimer $E_P(f)$, où P est la vraie distribution de probabilité

de la variable aléatoire X. Si Q est une approximation de P, d'après l'inégalité de Cauchy-Schwartz:

$$|\mathbb{E}_P(f) - \mathbb{E}_Q(f)| \leq ||P - Q||_2 \times ||f||_2$$

Ainsi, optimiser une fonction monotone de la norme L^2 entre P et Q (i.e. $||P-Q||_2$) semble l'approche naturelle.

Dans le cas où la structure courante du RB n'est pas la bonne, les deux approches ont des robustesses différentes. Le maximum de vraisemblance (approche "fréquentiste"[1] pour l'estimation de probabilité) implique des résultats instables; minimiser $||P-Q||_2$ offre alors une meilleure robustesse.

Ainsi la première contribution de cette partie est de proposer une fonction de coût non-standard (en apprentissage de RB), de montrer ses liens avec le critère L^2 et comment l'utiliser en pratique.

Une seconde contribution est une mesure de complexité pour les structures de RBs, prenant en compte un terme entropique structurel, en plus du classique nombre de paramètres du modèle. Ce nouveau critère vient avec des résultats théoriques qui montrent que:

- optimiser un compromis ad hoc entre cette mesure de complexité et l'erreur L^2 empirique conduit à une structure optimale en taille;

- la complexité de la structure d'un RB (définissant une famille de fonction: toutes les lois de probabilités ayant ces indépendances conditionnelles) est lié au terme entropique, permettant de distinguer des structures ayant le même nombre de paramètres.

- la mesure de complexité ne dépend que de la classe de distribution modélisée par la structure, i.e. on peut travailler sur les équivalents de Markov.

[1] L'approche fréquentiste pour estimer la probabilité d'un événement consiste simplement à compter combien de fois cet événement particulier apparait divisé par le nombre total d'apparition d'événements.

Résultats

Les approches paramétriques usuelles (section 2.3.1) conduisent asymptotiquement aux paramètres optimaux dans le cas où la structure du RB correspond aux indépendances conditionnelles de la distribution générant les données. L'avantage de cette méthode est qu'elle est très rapide, et les paramètres de chaque variable ne dépend que des variables locales (déterminer $P(B|A)$ ne dépend que de la fréquence des couples A,B). Une première contribution est de montrer cependant que cette approche est instable et non optimale pour le critère L^2 si la structure ne correspond pas à la décomposition de la loi jointe. Par opposition, la méthode proposée est plus chère en temps de calcul et basée sur un apprentissage global des paramètres mais est consistente (section 2.7.2).

Les bornes de risque montrent que la probabilité d'une estimation d'erreur plus grande qu'un certain ε est bornée par un certain δ dépendant de ε et du nombre d'exemples d'apprentissage. De façon équivalente, ces bornes donnent le nombre d'exemples nécessaires pour atteindre une erreur inférieure à ε avec probabilité au moins $1-\delta$.

Le cas des variables cachées (c'est à dire apparaissant dans le modèle, mais pas observables dans les exemples) est traité dans la section 2.6.5 en considérant le cas paramétrique et non-paramétrique.

Un algorithme avec convergence asymptotique vers l'erreur L^2 minimale est présenté en section 2.7.4 (Thm 8). De plus, nous démontrons la convergence de l'algorithme vers une structure minimale au sens défini par l'utilisateur (qui peut inclure les mesures classiques de complexité).

La comparison de notre mesure de complexité et les mesures usuelles fait apparaitre des termes principaux dans la mesure de complexité. Le *nombre de couverture* du RB associé à une structure donnée est directement lié à la complexité de la structure. La borne donnée par le théorème 7 dépend à la fois du nombre de paramètres R et de l'"entropie" $H(r)$ de la structure, où $H(r) = -\sum_{k=1}^{a} \frac{r(k)}{R} ln(\frac{r(k)}{R})$ et $r(k)$ est le nombre de paramètres de chaque noeud k ($R = \sum_k r(k)$). Il est montré de plus empiriquement que $H(r)$ est corrélé avec la complexité de la structure pour R fixé.

La section 2.8 présente différents algorithmes pour optimiser le critère proposé. Ces algorithmes sont basés sur une des méthodes quasi-Newton les plus standard, BFGS

[Bro70, Fle70, Gol70, Sha70], et utilisent une estimation non-triviale de la fonction de coût et de son gradient.

Enfin, l'approche est validée expérimentalement en section 2.9 en montrant sa pertinence statistique et son fonctionnement en pratique.

Programmation Dynamique Stochastique Robuste

Introduction

L'algorithme de programmation dynamique pour résoudre les PDMs est à la fois une des plus anciennes méthodes, mais aussi à la base de nombreux algorithmes plus récents. D'autre part, cet algorithme trouve sa place dans des applications d'optimisation en vraie grandeur dans lesquelles la robustesse est importante et où les conditions d'utilisation de cette méthode sont satisfaites (voir Chapitre 1).

Nous nous intéressons dans cette partie au cas continu, cas dans lequel chaque étape de l'algorithme pose un problème difficile en soit. Les étapes qui sont plus particulièrement étudiées dans cette thèse sont **l'optimisation** (une fois V_{t+1} connue, comment calculer le $\sup_{a\in\mathcal{A}}$), **l'apprentissage** (construire V_t à partir d'exemples (état, valeur)), et **l'échantillonnage** (comment choisir les exemples à partir desquels apprendre). Bien entendu, ces trois étapes sont interdépendantes mais ont aussi leur particularités propres.

Optimisation

Le $\sup_{a\in\mathcal{A}}$ est calculé de nombreuses fois pendant une résolution du problème par programmation dynamique. Pour T pas de temps, si N points sont requis pour approximer efficacement chaque V_t, alors il y a $T \times N$ optimisations. De plus, le gradient de la fonction à optimiser n'est pas toujours disponible, à cause du fait que des simulateurs complexes sont parfois employés pour calculer la fonction de transition. La fonction à optimiser est parfois convexe, mais souvent ne l'est pas. De plus, des mixtures de variables à la fois continues et discretes sont parfois à traiter.

Les exigences quand à l'optimisation sont principalement à propos de la robustesse. En optimisation non linéaire, elle peut avoir plusieurs définitions (on suppose qu'on cherche à

minimiser la fonction):

- Un premier sens est qu'une optimisation robuste est la recherche de x tel que la fonction a des valeurs faibles dans le voisinage de x, et pas seulement en le point x. En particulier, [DeJ92] a introduit l'idée que les algorithmes évolutionnaires ne sont pas des optimiseurs de fonction, mais des outils pour trouver de large zones où la fonction a des petites valeurs;

- un deuxième sens est l'évitement des optima locaux. Les méthodes déterministiques itératives sont souvent plus sujettes aux optimas locaux que par exemple des méthodes évolutionnaires; cependant, différentes formes de "restarts" (relancer l'optimisation d'un point initial différent) peut aussi être efficace pour éviter les optimas locaux;

- un troisième sens est la robustesse devant le bruit de la fonction à optimiser. Plusieurs modèles de bruits et études peuvent être trouvés dans: [JB05, SBO04, Tsu99, FG88, BOS04].

- une quatrième définition peut être la robustesse vis à vis de fonctions peu régulières, même s'il n'y a pas de minimum local. Des algorithmes qui sont basés sur le rang des valeurs des points visités (comme les algorithmes évolutionnaires) et non pas les valeurs elles-mêmes, ne dépendent pas, par construction, des transformations croissantes de la fonction objectif. Ce genre d'algorithmes sont optimaux vis à vis de ces transformations [GRT06]. Par exemple, $\sqrt{||x||}$ (ou des fonctions C^∞ proches de celle-ci) entraîne un très mauvais comportement des algorithmes de type Newton, comme BFGS, alors qu'un algorithme basé sur le rang a le même comportement sur $\sqrt{||x||}$ que sur $||x||^2$.

- un cinquième sens possible est la robustesse par rapport aux choix non déterministes de l'algorithme, ou en tout cas arbitraires (e.g. choix du point initial). Les algorithmes à base de population (conservant à chaque itération non pas un point candidat, mais plusieurs) sont plus robustes en ce sens.

Apprentissage

La regression dans le cadre de l'apprentissage par renforcement apporte des problématiques qui sont moins critiques dans le cadre de l'apprentissage supervisé classique. Les contraintes sont similaires à celles exposée pour l'optimisation, en particulier :

- De nombreuses étapes d'apprentissage ont a être effectuées, avec parfois un coût de calcul pour un exemple (qui demande de simuler au moins une étape de la transition, et plusieurs si une espérance doit être calculée) très élevé. Donc l'apprenneur doit être capable de travailler avec peu d'exemples dans certains cas;

- la propriété de robustesse de l'apprenneur est critique c'est à dire que le pire cas entre plusieurs apprentissages a plus de sens que l'erreur moyenne;

- l'existence de faux minima locaux dans la fonction d'approximation peut être problématique pour les optimiseurs;

- la fonction de coût qu'il faut appliquer lors d'un apprentissage (L^2, L^p, ...) n'est pas bien connu même en théorie (voir [Mun05]).

Echantillonnage

Comme noté par exemple par [CGJ95b], la capacité de l'appreneur de choisir ses exemples pour améliorer la qualité de l'apprentissage, est un point central dans l'apprentissage. Dans ce modèle de l'apprentissage, l'algorithme est fait de : (i) un échantillonneur qui choisit des points dans le domaine et (ii) d'un appreneur passif qui prend ces points et les labels. Nous distinguons dans la suite deux types d'apprentissage actif :

- Approches "aveugles" dans lesquelles les points sont choisis indépendamment de l'appreneur et des labels. Cela correspond essentiellement à des suites de points bien répartis dans le domaine par exemple de façon "quasi-aléatoire" [CM04].

- approches "non-aveugles", où l'échantillonnage utilise la connaissance des labels des points précédents et de l'appreneur pour choisir le ou les points suivants.

Il est à noter que bien que plus générales, la supériorité des méthodes non-aveugles n'est pas évidente. Notamment, explorer l'ensemble du domaine sans se focaliser sur une région qui semble intéressante permet d'éviter certaines erreurs.

L'étude de l'échantillonnage dans le cadre la PD est intéressante en soit par rapport au cadre de l'apprentissage supervisé, car le lien entre l'exploration et l'exploitation y est beaucoup plus fort.

Résultats

Les contributions principales de cette partie se déclinent en deux catégories. D'abord, le développement d'une plateforme opensource libre *OpenDP* offrant à la communauté un environnement commun et avec de nombreux outils pour comparer les algorithmes. Cette plateforme est utilisée pour effectuer une comparaison empirique de l'optimisation, apprentissage et échantionnage dans le cadre de la Programmation Dynamique (PD). D'autre part, des résultats théoriques à propos des méthodes d'échantillonnage et une nouvelle méthode d'échantillonnage active sont présentés.

Pour l'optimisation, les résultats expérimentaux démontrent comme attendu que les propriétés de robustesse exposées plus haut sont primordiales. Ce sont les algorithmes évolutionnaires qui ont souvent les meilleures performances. Plus précisement:

- **Gestion de stocks en haute dimension**. CMA-ES est un algorithme efficace quand la dimension augmente et pour des problèmes assez réguliers. Il est moins robuste qu'un algorithme évolutionnaire plus simple (appelé EA dans la thèse), mais apparait un très bon candidat pour l'optimisation non-linéaire nécessaire dans les problèmes (importants) de gestion de stocks en grande dimension, dans lesquels il y a assez de régularités. L'algorithme BFGS n'est pas satisfaisant : en PD avec approximation, la convexité et la dérivabilité ne sont pas des hypothèses réalistes, même si la loi des coûts marginaux s'applique. Les expériences se sont déroulées en dimension de 4 à 16, sans heuristique pour réduire la dimension, ou une réécriture du problème en dimension plus faible et les résultats sont clairement significatifs. Cependant, l'algorithme CMA-ES a un coût computationnel élevé. Il est donc adapté aux cas où la fonction à optimiser a un coût élevé, mais pourrait être moins intéressant dans les

cas où la fonction objectif a un coût de calcul négligeable.

- **Robustesse dans les cas peu réguliers.** Les algorithmes évolutionnaires sont les seuls à être plus efficaces que les optimisations par quasi-aléatoire de façon stable dans les problèmes peu réguliers. L'algorithme appelé "EA" n'est pas toujours le meilleur, mais il est la plupart du temps meilleur que l'optimisation par recherche aléatoire, ce qui est loin d'être le cas pour la plupart des autres algorithmes.

- **Un outil naturel pour les problèmes "bang-bang"**[2]. Dans certains cas (tipiquement les problèmes "bang-bang") la discrétisation LD, introduisant un biais vers les frontières, est de façon non surprenante la meilleure, mais est la pire sur d'autres problèmes. Ce n'est pas un résultat évident, car cela montre que la méthode LD fourni une façon naturelle de générer des solutions "bang-bang".

En apprentissage, l'algorithme SVM donne les meilleurs résultats pour un nombre important de problèmes, et aussi obtient de loin les résultats les plus stables. C'est cohérent avec les bons résultats de cet algorithme obtenus dans des benchmarks d'apprentissage supervisé. De plus, le coût en temps de calcul dans ces expériences est limité. Cependant, ce coût devrait augmenter rapidement avec le nombre d'exemples pour deux raisons. D'abord, le temps d'apprentissage peut être quadratique en le nombre d'exemples, et donc devenir prohibitif. L'autre source de coût en temps de calcul est moins souvent mis en avant dans l'apprentissage supervisé, mais a dans ce contexte une grande importance : le temps pour appliquer un appel de fonction. En effet, pour chaque état échantillonné, l'équation de Bellman appelle $N \times m$ fois la fonction V_{t+1}, avec m le nombre d'appels de fonction autorisé dans l'optimiseur, et N le nombre d'appel à la fonction de transition pour calculer l'espérance. Ainsi, avec n le nombre d'états échantillonnés et T l'horizon, la procédure de PD complète a besoin de $T \times n \times N \times m$ appels aux fonctions apprises. Le coût d'un appel à un V représenté par un SVM est linéaire en le nombre de vecteurs de support, soit en pratique dans les problèmes de regression, aussi linéaire en le nombre d'exemples. Avec

[2]Un problème est "bang-bang" quand l'action optimale se trouve sur une frontière de l'espace d'action. Pour un problème de conduite de voiture, cela voudrait dire que l'action optimale est d'accélérer à fond ou de freiner à fond.

$\alpha < 1$ ce coefficient de linéarité, la procédure prend environ $T \times \alpha n^2 \times N \times m$. Ainsi, le coût total des appels de fonction est aussi quadratique en le nombre d'exemples utilisés à chaque pas de temps, et ce terme est souvent dominant.

Les résultats de l'échantillonnage ne sont pas en compétition directe avec d'autres améliorations de l'apprentissage par renforcement, mais sont plutôt orthogonales. La partie théorique montre principalement qu'une part d'aléatoire est nécessaire, pour la robustesse, mais que cette part d'aléatoire peut être modérée et que les bonnes propriétés de vitesse de convergence des suites déterministes peuvent être conservées. Ces résultats sont à rapprocher du fait (voir par exemple [LL02]) que de l'aléatoire doit être ajouté dans les suites quasi-aléatoires pour plusieurs propriétés d'absence de biais.

Les conclusions expérimentales sont les suivantes :

- Notre nouvelle méthode d'échantillonnage appelée GLD est plus performante que tous les autres échantillonneurs aveugles dans la plupart des cas. En particulier c'est une extension naturelle pour tout les nombres de points et toute dimensionalité des échantillonnages par grille régulières, et est différente et meilleure que les autres approches à faible dispersion. Cette approche aveugle est la plus stable (première dans la plupart des benchmarks); cependant, ses faiblesses se révèlent lorsque les frontières ne sont pas pertinentes.

- Notre nouvelle méthode d'échantillonnage non aveugle est facile à utiliser ; il est facile d'y ajouter de la connaissance experte (augmenter α pour explorer plus le domaine, le réduire pour se concentrer sur les parties optimistes, changer le critère, ajouter de la diversité à l'algorithme évolutionnaire). Cependant, les expériences montrent que le critère testé ici n'est pas universel ; il fonctionne pour le problème "many-bots", donnant des stratégies avec de meilleures performances que toutes les techniques aveugles testées, mais obtient de mauvais résultats pour d'autres problèmes. Ceci est plutôt décevant, mais nous pensons que les méthodes d'échantillonnage non-aveugles demandent en général de la mise au point fine. L'algorithme, en changeant le paramètre α passe continument de l'échantillonnage aléatoire pur (ou potentiellement quasi-aléatoire, cela n'a pas été testé ici), jusqu'au

sur-échantionnage des zones vérifiant une heuristique spécifiée par l'utilisateur. Un des avantages de cette méthode est qu'elle semble fonctionner en dimension raisonnablement grande, ce qui est un cas rare pour les méthodes d'apprentissage actif, spécialement en regression.

Cas discret de grande dimension: Computer-Go

Introduction

Les jeux sont traditionnellement une source d'applications pour le contrôle optimal discret et l'Apprentissage par Renforcement. Certains algorithmes d'AR (par exemple $TD(\lambda)$ et ses variantes) ont atteint un niveau de maître dans le jeu de dames [SHJ01a], Othello [Bur99], et les échecs [BTW98]. Le Go et le Poker restent deux jeux dans lesquels les programmes ne peuvent inquiéter les meilleurs humains.

Beaucoup d'approches basées sur l'AR contiennent : i) une exploration par structure d'arbre de l'espace d'états ; ii) une fonction de valeur, pour évaluer les feuilles et les noeuds de l'arbre. Le jeu de Go, un jeu Asiatique plusieurs fois millénaires qui jouit d'une grande popularité à travers le monde, impose des challenges dans ces deux directions. D'un côté, même si les règles sont simples (voir la figure 1), l'espace d'états est très grand et le nombre moyen de coups possibles (et donc le facteur de branchement de l'arbre) est supérieur à 200. Au contraire, les règles des échecs sont plus complexes mais le nombre moyen de coups possibles est environ 40. D'un autre côté, il n'y a pas de solution réellement efficace pour modéliser et apprendre la fonction de valeur. Ainsi, l'algorithme alpha-beta ne donne pas de bon résultats, contrairement aux échecs où par exemple DeepBlue [New96] dépend essentiellement de alpha-beta. Ces deux raisons expliquent pourquoi le Go est considéré comme un des grands challenges de l'intelligence artificielle, remplaçant les échecs dans ce rôle [BC01]. Le tableau 1 montre le status actuel entre les meilleurs programmes et les meilleurs humains dans plusieurs jeux célèbres.

Des progrès récents sont apparus pour l'évaluation des positions basée sur des approches par Monte-Carlo [Bru93a]. Cependant, cette procédure d'évaluation a une précision limitée ; jouer le coup avec le plus grand score dans chaque position ne donne pas

Jeu	Année du premier prog. *Nom du prog.* (Auteur)	Niveau contre les humains	Date victoire et/ou *Nom du prog.*
Echecs	1962 *Kotok-McCarthy*	$M > H$	1997 *Deeper Blue*
Backgammon	1979 *SKG 9.8* (Berliner)	$M \geq H$	1998 *TD-Gammon*
Checkers	1952 (Strachey, Samuel)	$M >> H$ $\approx$ *resolu*	1994 *Chinook*
Otello	1982 *IAGO* (Rosenbloom)	$M >> H$	1997 *LOGISTELLO*
Scrabble	1977 (Shapiro and Smith)	$M > H$	1998 *MAVEN*
Bridge	1963 (Berlekamp)	$M \approx H$	*GIB*
Poker	1977 (Findler)	$M << H$	*POKI*
Go	1970 (Zobrist)	$M << H$	*MoGo*

Table 1: Status du niveau des meilleurs programmes contre les meilleurs humains dans plusieurs jeux populaires. La première colonne correspond aux noms des jeux. La seconde colonne est l'année du premier programme jouant à ce jeu à un niveau intéressant, avec parfois le nom du programme et le nom de l'auteur. La troisième colonne donne le status actuel entre le meilleur programme (M) et le champion du monde (H). Si le programme est meilleur que l'humain, alors la dernière colonne donne l'année et le nom du programme ayant battu la première fois le champion du monde. Sinon, la dernière colonne donne le nom du meilleur programme actuel.

la victoire. Une approche exploration/exploitation, basée sur le problème du bandit manchot est considérée. Dans ce modèle, plusieurs machines à sous sont disponibles, chacune avec une certaine probabilité de victoire. Le joueur doit maximiser la somme totale de ses gains [ACBFS95]. L'algorithme UCB1 [ACBF02] a été récemment étendu à un espace structuré en arbre par Kocsis et Szepesvari (algorithme UCT) [KS06].

Les contributions de cette partie peuvent se décliner en deux ensembles. Le premier concerne la politique de simulation utilisée dans l'évaluation par Monte-Carlo. Le second concerne l'arbre de recherche par l'algorithme UCT.

Résultats

La politique de simulation

La méthode de Monte-Carlo permet d'approximer l'espérance de gain à partir d'une position, si on suit une certaine politique (de jeu). Cette approximation devient plus précise

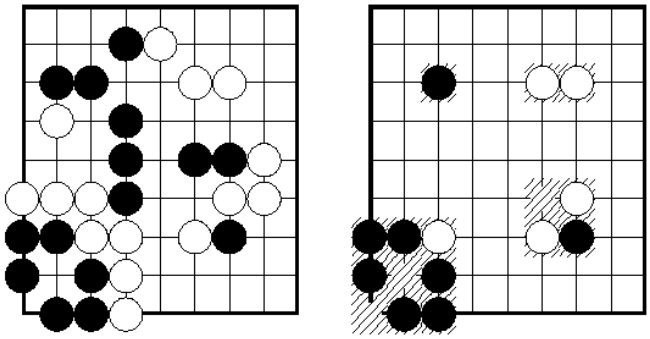

Figure 1: (Gauche) Une position d'exemple pour le jeu de go sur plateau 9×9. Le noir et le blanc jouent alternativement pour placer des pierres. Les pierres ne peuvent pas jamais bouger mais peuvent être capturées si elles sont complètement entourées. Le joueur qui entoure le plus de territoire a gagné la partie. (Droite) Les formes ont une place importante dans la strategie du jeu de go. La figure montre (dans le sens des aiguilles d'une montre en partant d'en haut à gauche) des formes locales de taille 1×1, 2×1, 2×2 and 3×3 qui apparaissent dans la position de l'exemple.

à mesure que le nombre de simulation augmente, et converge vers l'espérance, mais cette espérance n'est qu'une approximation de la "vraie" valeur min-max de la position.

Nos contributions pour la politique de simulation se déclinent sous trois axes. Le premier est l'utilisation de séquences de coups forcés dans les simulations, grâce à des motifs locaux. Cette modification augmente le niveau de jeu de façon très significative. D'autre part, des résultats surprenants sont apparus : une politique de simulation qui a, comme joueur, un niveau plus élevé, ne donne pas forcément de meilleurs résultats lorsqu'utilisé dans Monte-Carlo. Nous utilisons notamment une politique dérivée d'une fonction de valeur apprise par l'algorithme $TD(\lambda)$ et une approximation linéaire. Une troisième contribution correspond à quelques éléments théoriques montrant qu'une mémoire permet de découpler le niveau de jeu d'une politique avec la précision de l'estimation par Monte-Carlo.

Comprendre plus avant le lien entre politique et précision de l'estimation pourrait être la clé des avancées supplémentaires.

L'arbre de recherche

Le second ensemble de contributions correspond à la partie de l'algorithme de l'arbre de recherche, UCT. Plusieurs améliorations sont proposées, la principale étant l'algorithme "RAVE" (Rapid Action Value Estimation), qui est inspiré de l'heuristique "All moves as first" dans le computer go et est adaptée à UCT. Cet algorithme apporte plusieurs centaines de points ELO d'amélioration du niveau de jeu [3] particulièrement quand seulement un petit nombre de simulations par coup est disponible [4]. Mais encore plus significativement, cela permet d'atteindre un niveau relativement élevé sur des plateaux de jeu plus grands, qui posent beaucoup plus de problèmes aux algorithmes basés sur Monte-Carlo/UCT. L'idée est de considérer l'ensemble des coups joués par la politique dans une simulation Monte-Carlo, et, dans une position donnée, en plus d'estimer la probabilité de gagner sachant qu'on a effectué une action a au premier coup), on estime aussi la probabilité de gagner sachant qu'on effectue a avant l'adversaire (mais pas forcément au premier coup). Ceci peut être directement appliqué à d'autre problèmes où certaines actions peuvent permuter.

Conclusion et Perspectives

Cette thèse présente plusieurs contributions dans le domaine de l'apprentissage par renforcement, ou domaines connexes.

La première partie (Chapitre 2) présente un modèle très populaire, les Réseaux Bayesiens, cette famille de modèles graphiques pouvant tirer avantage de la structure du problème, représentant par exemple un PDM factorisé à l'aide d'un Réseau Bayésien Dynamique, permettant ainsi de traiter des problèmes de plus grandes tailles. Tout d'abord, il est montré que l'apprentissage peut être fait en utilisant une fonction de coût L_2 plutôt que le classique maximum de vraisemblance. Le premier avantage est que l'erreur d'approximation d'une espérance calculée à partir d'un modèle est directement liée à cette mesure L_2 de l'erreur d'apprentissage. Le deuxième avantage est la convergence vers les

[3] Le classement ELO (`http://en.wikipedia.org/wiki/Elo_rating_system`) est un système de classement où la probabilité de victoire est directement liée à la différence entre les rangs.

[4] Une version de MoGo limitée à seulement 3000 simulations par coup, i.e. jouant en moins d'une seconde chaque coup a atteint un niveau de 2000 ELO sur le server "Computer Go Online Server"

paramètres optimaux (à structure fixée) lors de l'apprentissage à partir d'une structure erronnée. Cependant, l'optimisation des paramètres est computationellement beaucoup plus coûteuse que la méthode fréquentiste, et nous proposons des algorithmes pour rendre l'apprentissage possible en pratique. D'autre part, nous démontrons des bornes sur l'erreur d'apprentissage. A partir de ces bornes, est dérivé un nouveau score pour les structures et nous montrons sa pertinence expérimentalement. Nous utilisons aussi ces résultats pour proposer des algorithmes d'apprentissage basés sur les théorèmes.

La seconde partie s'intéresse à la Programmation Dynamique dans le cas d'espaces d'états et d'actions continus. La première contribution est notre plateforme OpenDP, pouvant être utilisée par la communauté comme un outil pour comparer les algorithmes sur différents problèmes. La seconde contribution est une étude expérimentale sur l'effet des algorithmes d'optimisation non-linéaires, l'apprentissage par régression et l'échantillonnage, dans le cadre de la programmation dynamique en continu. Dans ces trois étapes, la robustesse de l'algorithme est mise à l'épreuve pour obtenir un contrôle de qualité après intégration. Dans ces tâches, les algorithmes évolutionnaires, les SVMs, et les suites à faible dispersion donnent les meilleurs résultats dans respectivement l'optimisation, la regression et l'échantillonnage. La troisième contribution correspond à quelques éclairages théoriques sur l'échantillonnage, l'aspect stochastique qui doit être conservé, et une nouvelle méthode d'échantillonnage orthogonale aux autres méthodes d'échantillonnage améliorées en programmation dynamique.

La troisième partie traite un problème de contrôle en haute dimension, le jeu de go. Ce problème est un grand challenge actuel pour l'apprentissage informatique. Les méthodes présentées pourront potentiellement être aussi appliquées à d'autres domaines. Les contributions de ce châpitre gravitent autour de l'algorithme UCT, son application au jeu de go, puis des améliorations dans la recherche arborescente et la politique de simulation permettant d'évaluer les positions. Ces contributions ont été appliquées dans notre programme MoGo[5]. La première contribution concerne l'introduction de séquences de coups forcés grâce à l'application de motifs dans la politique de simulation, permettant une évaluation beaucoup plus précise des positions. Nous montrons cependant

[5]MoGo a gagné plusieurs tournois internationaux dans toutes les tailles courantes de goban, avec de plus des résultats positifs contre des joueurs humains forts notamment dans les petites tailles de goban, et est le programme le plus performant au moment de l'écriture de cette thèse.

expérimentalement que la qualité de la politique de simulation en tant que fonction d'évaluation n'est pas directement liée à son efficacité en tant que joueur. Nous proposons quelques explications sur le plan théorique de ce résultat surprenant. La seconde contribution principale concerne la généralisation des valeurs des actions à l'intérieur de l'arbre de recherche. Cette généralisation augmente significativement les performances dans les petites tailles de goban, mais surtout permet à l'algorithme de se montrer efficace même dans les plus grandes tailles de goban.

Après avoir présenté à la fois un module d'apprentissage de modèle et un module de prise de décision, une extension naturelle de ces travaux serait de les combiner dans une application grandeur nature. Le jeu de poker propose un autre problème bien défini mais difficile dans lequel, contrairement au jeu de go, un modèle de l'adversaire est nécessaire. L'incertitude sur l'adversaire représente l'incertitude sur la fonction de transition du PDM sous-jacent (les règles du jeu sont bien entendu supposées connues). Un stage en cours dans notre groupe de recherche combine un arbre de recherche basé sur Monte-Carlo dans le jeu Texas Hold'em à deux joueurs. L'adversaire peut être modélisé par un réseau bayésien, utilisé comme outil pour approximer la probabilité de chaque action dépendant de variables observables et cachées. Le critère d'apprentissage proposé dans le châpitre 2 est alors approprié, puisqu'on s'intéresse à l'espérance du gain.

Etant donné que les problèmes rééls ont une structure sous-jacente, une des directions de recherche les plus prometteuses pour les algorithmes basés sur UCT est de généraliser les données entre les noeuds de l'arbre. L'idée serait d'apprendre en ligne une approximation des valeurs des états spécialisée pour la distribution des états induite par UCT. Ainsi, l'approximation est effective seulement pour un très petit sous-ensemble de tous les états possibles, permettant une meilleure utilisation de la capacité d'approximation de l'approximateur de fonctions que s'il était employé sur tout l'espace d'états.

Une autre direction de recherche est la parallélisation massive de l'algorithme UCT sur un grand ensemble de machines. Bien que des implémentation sur des machines multi processeurs à mémoire partagée ont été couronnées de succès, des études théoriques et pratiques du comportement de l'algorithme dans le cadre de communications inter-machines plus coûteuses pourrait bénéficier à des applications à grande échelle. La généralisation

entre noeuds de l'arbre précédemment évoquée pourrait alors être utilisée pour minimiser les données à échanger entre machines.

Cette thèse met l'accent sur les interactions de l'apprentissage par renforcement avec d'autres domaines de recherche comme l'apprentissage supervisé, l'apprentissage non-supervisé, l'optimisation dans le discret ou le continu, l'intégration numérique et la théorie de l'apprentissage. Nous avons aussi mis l'accent sur l'importance d'applications grandeur nature, et exploré comment des algorithmes simples pouvaient passer à l'échelle sur des problèmes de plus grande taille. Nous pensons que tous ces éléments peuvent être intégrés ensemble et que la performance d'un tel système pourraient être meilleure que la somme de ses parties.

PHD THESIS

by

SYLVAIN GELLY

A CONTRIBUTION TO REINFORCEMENT LEARNING; APPLICATION TO COMPUTER-GO

Université Paris-Sud, 91405 Orsay Cedex, France

Acknowledgements

While there is only one author on this thesis, this work would never have been done without the help of many people on different aspects. I first want to thank my referees Rémi Munos and Csaba Szepesvari who kindly accepted to read this thesis and the members of my jury Jacques Blanc-Talon, Olivier Bousquet and John Shaw-Taylor. The rest of those acknowledgements are written in french, my native language, but I thank any reader whatever his language is :).

Je remercie d'abord par ordre chronologique Marc Schoenauer que j'ai rencontré lors d'un module à l'Ecole Polytechnique, et qui m'a alors proposé un stage d'option. C'est à ce moment que j'ai rencontré Michèle Sebag qui a encadré ce stage, mais aussi mon stage de Master, et bien entendu ma thèse, et que je remercie donc pour ces presques 4 années de travail en commun. Merci aussi à Nicolas Bredeche qui m'a introduit aux problématiques de robotique co-encadrant mon stage de master et finalement ma thèse.

Un grand merci tout particulier à Olivier Teytaud, arrivé dans notre équipe au début de ma thèse, avec qui j'ai beaucoup travaillé. J'ai profité et appris non seulement de ses connaissances et compétences allant de la théorie la plus abstraite à l'application d'envergure, mais aussi de son indéfectible énergie sans cesse renouvelée.

Un deuxième merci particulier pour Yizao Wang, qui a introduit le sujet de la programmation du jeu de go dans l'équipe, et avec qui j'ai passé un été 2006 plein de motivation, d'espoirs tantôt déçus, tantôt réalisés, travaillant jour et nuit sur notre programme MoGo. Dommage que tu ne sois pas resté dans l'équipe... Promis Yizao, un jour j'apprendrai le (trois mots de) chinois :-).

Merci à tous les doctorants ou ex-doctorants de l'équipe (ou proches), s'entraidant dans l'adversité. Dans l'ordre (pseudo-ou-pas-du-tout-)chronologique, merci à Jérémie (bientôt

à Las Vegas), Nicolas (on parie un coca sur csmash?), Yann (ce n'est qu'une histoire de bulle), Anne (maintenant tu es une grande!), Mohamed (le C++ est ton ami), Mary (les robots nous remplaceront un jour), Cédric (un volley un jour?), Celso (vrai joueur de volley lui), Engueran (encore du volley!), Mathieu (même pas peur de tes ogres), Thomas (non pas de piscine), Raymond (à quand un nouveau Mare Nostrum?), Alexandre (l'oeuf, la poule et les robots), Fey (tu peux demander maintenant), Damien (bientôt un petit nouveau), Romaric (battre en retraite en avançant c'est pas juste!), Arpad (dommage pas le temps d'apprendre à jouer au go), Julien (+10% c'est beaucoup), Ettore (le pays basque c'est mieux que Strasbourg), Miguel (citoyen européen tu as résolu le problème de la langue unique: parler toutes les langues!), Cyril (du trafic routier aux BCI), Xiangliang (Zen me yang?), Lou (bravo pour la journée de plage à Marseille avec ordre de mission!), et bien sûr tous les autres que je ne mentionne pas mais à qui un merci est bien sûr adressé et qui me pardonneront. Bien que non membre de l'équipe, merci à David Silver, m'invitant à la dernière minute :-) à Edmonton, pour toutes les collaborations et discussions fructueuses (et aussi celles non fructueuses). Espérons que les dernières modifications sur cette thèse, le dernier soir, au chant des cigales, feront partie des fructueuses :-).

Bien entendu, les remerciements les plus chaleureux vont en direction de ma famille qui m'a toujours soutenu dans mes choix, et surtout ma mère et mon frère. Arnaud, pendant ces années que j'ai passées en région parisienne tu as pu t'entrainer à la planche, mais je vais me rattraper !

Last but not least, merci à ma compagne Elodie, qui m'a supporté tous les jours depuis tant d'années, soutenus dans les moments difficiles, sans qui, plus que tout autre, cette thèse n'existerait pas.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reinforcement Learning (RL) [SB98, BT96] is at the interface of control theory, supervised and unsupervised learning, optimization and cognitive sciences. While RL addresses many objectives with major economic impact (e.g. marketing [AVAS04], power plant control [SDG+00], bio-reactors [CVPL05], Vehicle Routing [PT06], flying control [NKJS04], finance [MS01], computer games [SHJ01b], robotics [KS04]), it raises deep theoretical and practical difficulties.

Formally, RL considers an *environment* described from *transitions*; given the current *state* of the system under study, given the current *action* selected by the controller, the transition defines the next *state*. The environment also defines the *reward* collected (or the cost undergone) by the system when going through the visited states. The goal of RL is to find a *policy* or controller, associating to each state an action in such a way that it optimizes the system reward over time. RL proceeds by learning a policy through exploring the state $\times$ action space, either actually (*in situ*) or virtually (*in silico*, using simulations).

This manuscript presents the thesis contributions to the field of Reinforcement Learning:

- Chapter 2 is devoted to Bayesian Networks (BNs), which are commonly used as a representation framework for RL policies. Our theoretical contribution is based on a new bound for the covering numbers of the BN space, involving the network *structural entropy* besides the standard number of parameters. The algorithmic contribution regards the parametric and non parametric optimization of the learning criteria

based on this bound, the merits of which are empirically demonstrated on artificial problems.

- Chapter 3 investigates the Stochastic Dynamic Programming (SDP) framework, focussing on continuous state and action spaces. The contributions regard non-linear optimization, regression learning and sampling through empirical studies. Those studies are all conducted in our framework OpenDP, a tool to compare algorithms in different benchmark problems.

- Chapter 4 focuses on a high-dimensional discrete problem, Computer-Go, which is commonly viewed as one of the major challenges in both Machine Learning and Artificial Intelligence [BC01]. A new algorithm derived from the Game Theory field, called UCT [KS06] is adapted to the Go context. The resulting approach, the MoGo program, is the world strongest Go program at the time of writing, and opens to exciting perspectives of research.

The rest of this chapter is meant to introduce the scope of the thesis and provide the general formal background for the other chapters. Specific bibliography will also be provided for each chapter.

1.1 Markov Decision Process (MDP)

This section presents Markov Decision Processes (MDP), the most widely used framework for Reinforcement Learning. Specific flavors of MDP will be introduced thereafter, namely Model-free (section 1.2) and Model-based (section 1.3) approaches.

The MDP framework describes a control problem where an *agent* evolves in an *environment* and aims to optimize some *reward*. As an example, a car driver (the agent) deals with the physics laws, the map and the traffic (the environment); the agent's goal is to minimize his time to destination while avoiding accidents. In order to do so, the agent applies a *policy* (e.g. turn left and right, speed up, up to the speed limit). The success of a given policy is assessed from the associated reward. Learning a good policy is the goal of RL algorithms, and the most common RL algorithms will be presented in this section.

Formally, a MDP is a tuple $(\mathcal{S}, \mathcal{A}, p, r)$ where:

- $\mathcal{S}$ is the state space (which can be discrete or continuous);
- $\mathcal{A}$ is the action space (which can be discrete or continuous);
- p is the probability transition function (defined below);
- r is the reward function (defined below).

While the time domain can be either discrete or continuous, the discrete time framework will be considered in the rest of this chapter for the sake of simplicity; meanwhile, many real-world applications can be described within the discrete time framework.

The probability transition function p_t is defined at every time t from $\mathcal{S} \times \mathcal{S} \times \mathcal{A}$ onto $\mathbb{R}^+$. More precisely, for a given time t, a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, $p_t(.|s,a)$ is the probability distribution on $\mathcal{S}$ describing the subsequent states one can reach taking action a in state s. In particular[1]

$$\forall t,\ \forall s \in \mathcal{S},\ \forall a \in \mathcal{A}, \int_{s' \in \mathcal{S}} p_t(s'|s,a)ds' = 1$$

Remark: *In the case where $\mathcal{S}$ is continuous, $p_t(s'|s,a)$ must be interpreted as a density rather than a probability (i.e. $p_t(s'|s,a) \neq P(s'|s,a)$; $\int_{s' \in S} p_t(s'|s,a)ds' = P(s' \in S|s,a)$). Along the same lines, in the case where the transition is deterministic $p_t(s'|s,a)$ might become a Dirac distribution. For the sake of simplicity, in the rest of the chapter the distinction between probabilities and probability density function will not be emphasized unless necessary.*

The reward function r_t is similarly defined at every time t from $\mathcal{S} \times \mathcal{A}$ onto $\mathbb{R}$. More precisely, for a given time t, a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, $r_t(s,a)$ is the instantaneous reward gathered by the agent taking the action a in state s at time t. Note that $r_t(s,a)$ can be stochastic (a random variable) too.

By definition, the MDP is said to be *stationary* if the transition and reward functions do

[1] While the summation is denoted by an integral ($\int$) for the sake of generality, it obviously corresponds to a sum ($\sum$) when considering a discrete state space.

not depend on time t, that is:

$$\exists p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R},\ \forall t\ p_t = p$$

and

$$\exists r : \mathcal{S} \times \mathcal{A} \to \mathbb{R},\ \forall t\ r_t = r$$

The main property of MDPs is the *Markov property*, namely the fact that the transition and reward functions only depend on the current state and action, and not on the system history (its past states and actions). Formally, a *history* denotes a (finite) sequence of (state, action) pairs: $h_t = (s_0, a_0, ..., s_{t-1}, a_{t-1})$. Noting H_t the set of all possible histories at time t, the Markov property is defined as:

$$\forall t,\ \forall h_t \in H_t,\ \forall s_t, s_{t+1} \in \mathcal{S},\ \forall a_t \in \mathcal{A},\ P(s_{t+1}|h_t, s_t, a_t) = P(s_{t+1}|s_t, a_t) = p_t(s_{t+1}|s_t, a_t)$$

Remark: *In some cases, i) the state space $\mathcal{S}$ and the action space $\mathcal{A}$ can depend on the current time step; the action space $\mathcal{A}$ can depend on the current state. For the sake of simplicity and with no loss of generality, we shall note $\mathcal{S}$ and $\mathcal{A}$ in the following instead of $\mathcal{S}(t)$ and $\mathcal{A}(t,s)$.*

1.1.1 Policy

A core notion of Reinforcement Learning, the *policy* models the agent's decisions. A policy is a function π associating to every time step t and each state s in $\mathcal{S}$, a distribution over actions in $\mathcal{A}$. Policy π is called *deterministic* if $\forall t,\ \forall s \in \mathcal{S}, \pi_t(s)$ is a Dirac distribution on $\mathcal{A}$. We then note $\pi_t(s) \in \mathcal{A}$ by abuse of notation.

Likewise, policy π is called *stationary* when it does not depend on the time step t, i.e. $\exists \pi,\ \forall t,\ \forall s \in \mathcal{S}\ \pi_t(s) = \pi(s)$.

Remark: *While the agent's decisions could depend on the whole system history, thanks to the Markov property, for any policy π there exists a policy π' which only depends on the current state, and which has the same value function (see 1.1.3) as π. History-dependent policies will therefore be discarded in the following.*

1.1.2 Criteria

Resolving an MDP problem proceeds by optimizing some criterion, the most standard of which are described in the following. In many applications the goal can be described naturally as a sum of instantaneous rewards.

Remark: *While every criterion in the following obviously depends on the current policy, only the rewards r_t will be involved in the analytic formulas below; the r_t are viewed as random variables depending on the actual policy.*

Finite horizon

In this case, the time interval is limited, e.g. the time to perform the task at hand is known in advance. Noting T the maximum number of time steps allowed to perform the task (horizon), then the criterion to maximize is the expectation of the sum of rewards over all time steps:

$$E[\sum_{t=0}^{T-1} r_t]$$

γ-discounted

When the task is not naturally limited wrt time, another very natural criterion proceeds by exponentially discounting the rewards of future time steps. This discount results in nice convergence properties (thanks to the contraction properties of the operators, see below); its interpretation is also very intuitive in economical terms, where the discount $\gamma \leq 1$ reflects a positive interest rate (e.g. current reward $(1+\varepsilon)r_t$ corresponds to r_{t+1} with interest rate $\varepsilon = 1/\gamma - 1$).

Denoting $\gamma \in [0,1)$ the discount factor, the reward is defined as:

$$E[\sum_{t=0}^{\infty} \gamma^t r_t]$$

This criterion also defines a "smooth horizon", $T = \sum_{t=0}^{\infty} \gamma^t = \frac{1}{1-\gamma}$.

Total

As a limit case of the γ-discounted criterion when $\gamma \to 1$, the total criterion is defined as:

$$\lim_{T\to\infty} E[\sum_{t=0}^{T-1} r_t]$$

Clearly, this criterion is relevant iff it is not infinite, e.g. when the reward is zero after some time step (which might depend on the policy).

Average

In some cases, we are only interested in the average reward, more specifically in the long term average (discarding the beginnings of the history). Formally, the average reward is defined as:

$$\lim_{T\to\infty} \frac{1}{T} E[\sum_{t=0}^{T-1} r_t]$$

In the following, we focus on the finite horizon and the γ-discounted criterions, as these are the most commonly used on the one hand, and related to our applicative contexts on the other hand.

1.1.3 Value functions

The crux of the RL problem is the value function $V^\pi(.)$ associated to a policy π; the value function is used to compare different policies, and it plays a key role in many algorithms. Several definitions, related to the above criteria (finite horizon, γ-discounted, total and average), are considered:

In the finite horizon case,

$$V^\pi(s) = E^\pi \left[\sum_{t=0}^{T-1} r_t | s_0 = s \right]$$

with E^π the expectation of realizations of MDP when following policy π. We will use E for short when there is no ambiguity.

Similarly, for the γ-discounted criterion:

$$V_{\gamma}^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$$

One can define the value function starting at any time step. For the finite horizon case, it becomes:

$$V_{t,T}^{\pi}(s) = E\left[\sum_{t'=0}^{T-1} r_{t'} | s_t = s\right]$$

and for the γ-discounted:

$$V_{t,\gamma}^{\pi}(s) = E\left[\sum_{t'=0}^{\infty} \gamma^{t'} r_{t'} | s_t = s\right]$$

Similar definitions are provided for the other criterions.

Let $\mathcal{V}$ denote the set of all possible value functions for a given criterion.
Equivalently, the value function can be defined in terms of Q values, where

$$Q_t^{\pi}(s,a) = r_t(s,a) + \gamma \int_{s' \in \mathcal{S}} p_t(s'|s,a) V_{t+1}^{\pi}(s')$$

Both definitions are equivalent ($V^{\pi}(s) = E_{a \sim \pi(s)} Q^{\pi}(s,a)$), with different strengths and weaknesses in practice.

The major merit of Q functions is to enable action selection in a direct way, based on the "only" optimization of Q whereas the use of V requires the additional use of the transition function and the expectation computation.

In contrast, V is defined on $\mathcal{S}$ (whereas Q is defined on $\mathcal{S} \times \mathcal{A}$). The domain size can be then much smaller.

In the following, the algorithms will be described using either the V or the Q functions, preferring the simpler or most common description.

1.1.4 Algorithms

With no loss of generality, let us consider the goal of maximizing the chosen criterion, and let us denote π^* the optimal policy (which exists, see below):

$$\forall V \in \mathcal{V},\ \forall \pi,\ \forall s \in \mathcal{S},\ V(s) = V^{\pi}(s) \leq V^{\pi^*}(s) = V^*(s)$$

Note that finding π^* or the associated value function V^* is equivalent in principle; once π^* is given, V^* is computed thanks to the definition of V^{π^*}, and if V^* is given, then[2].:

$$\forall s \in \mathcal{S}\ \pi^*(s) \in \sup_{a \in \mathcal{A}} \left(r_t(s,a) + \gamma \int_{s' \in \mathcal{S}} p_t(s'|s,a) V^*(s') ds' \right)$$

In practice, V^* can only be known up to some approximation (especially in the continuous case) and the $sup_{a \in \mathcal{A}}$ is also an approximation in the continuous case (see section 3.3).

A key notion for solving MDP is the Bellman operator [Bel57], noted L_t (or L if the problem is stationary) .

$$\forall t,\ \forall V \in \mathcal{V},\ \forall s \in \mathcal{S},\ (L_t V)(s) = \sup_{a \in \mathcal{A}} \left(r_t(s,a) + \int_{s' \in \mathcal{S}} p_t(s'|s,a) V(s') ds' \right)$$

For the γ-discounted criterion, it becomes:

$$\forall t,\ \forall V \in \mathcal{V},\ \forall s \in \mathcal{S},\ (L_{\gamma,t} V)(s) = \sup_{a \in \mathcal{A}} \left(r_t(s,a) + \gamma \int_{s' \in \mathcal{S}} p_t(s'|s,a) V(s') ds' \right)$$

Dynamic Programming (finite horizon)

Let us focus on the finite horizon and discrete time setting. A standard algorithm (referring the interested reader to [Ber95, BT96] for a general introduction, and to [SB98] for many extensions including Real Time Dynamic Programming (RTDP)), proceeds by building the value function and learning an optimal policy, assuming that the MDP is either known or modelled. The principles of this algorithm are furthermore the core of several classical

[2]Holds in the stationary case. If the MDP is non stationary, the equation remains the same but one must know V_t^* for each time step t

algorithms, described later on.

Indeed the Bellman's optimality principle [Bel57] states that the value function of the optimal policy at time step t is sufficient to build the value function of the optimal policy at time step $t-1$. As stated before, the value function of the optimal policy is also sufficient to build the optimal policy (except for the precision on the optimization step, i.e. computation of $\sup_{a\in\mathcal{A}}$).

Clearly the optimal action at the final time step T is the one which maximizes reward r_T; therefore the value function of the final time step is easy to compute. By induction, thanks to a backward through time computation, dynamic programming can be used to compute the value function of the optimal policy V^* for every time step, and thus the optimal policy π^* itself.

Algorithm 2 describes the dynamic programming algorithm when $\mathcal{S}$ and $\mathcal{A}$ are finite; other cases (where the spaces are infinite or highly dimensional) will be detailed in the following.

Algorithm 2 Dynamic Programming in finite state and action spaces

Input: a MDP, with $\mathcal{S}$ and $\mathcal{A}$ finite, finite horizon T.
Initialize $\forall s \in \mathcal{S},\ V_T(s) = 0$
for $t = T-1$ **backwards to** 0 **do**
  for all $s \in \mathcal{S}$ **do**
    $V_t(s) = L_{t+1}V_{t+1}(s)$
  end for
end for
Output: $\forall t,\ V_t = V_t^*$.
Output: $\forall t, \forall s\ \pi_t(s) = \arg\max_{a\in\mathcal{A}} \left(r_t(s,a) + \int_{s'\in\mathcal{S}} p_t(s'|s,a)V_{t+1}(s')ds'\right) = \pi_t^*(s)$.
Complexity: $T \times |\mathcal{S}|^2 \times |\mathcal{A}|$.

Although the argmax (in $\pi_t(s) = \arg\max_{a\in\mathcal{A}} \left(r_t(s,a) + \int_{s'\in\mathcal{S}} p_t(s'|s,a)V_{t+1}(s')ds'\right)$) is not uniquely defined, picking any action in the argmax set still leads to an optimal policy.

When the state space $\mathcal{S}$ is continuous or involves a large number of states, the above algorithm faces a computational challenge, regarding:
A. The estimation of the value $V_t(s)$ for each state $s \in \mathcal{S}$.
B. The expectation $\int_{s'\in\mathcal{S}} p_t(s'|s,a)V_{t+1}(s')ds'$.

Symmetrically, if the action space $\mathcal{A}$ is continuous or highly dimensional, the computation of $\sup_{a\in\mathcal{A}}$ and $\arg\max_{a\in\mathcal{A}}$ of algorithm 2 becomes computationally heavy (see section 3.3).

Regarding the latter issue, i.e. computing the expectation $\int_{s'\in\mathcal{S}} p_t(s'|s,a)V_{t+1}(s')$, we will use the term *random processes* to name the model of the transition function p_t, used to compute the value expectation itself.

Stochastic dynamic programming heavily relies on such random processes. The expectation may be computed exactly using prior knowledge, e.g. if the physics of the problem domain provides the transition model with a finite support. Monte-Carlo approaches can be used based on an existing simulator (simulating several transitions and taking the empirical average). Lastly, the transition function can be learned from data, e.g. using a dynamic bayesian network (see Chapter 2). In the remainder of this chapter, we only assume the existence of a transition model allowing the approximation/computation of the expectation involved in the value function.

Regarding the optimization step (selection of the optimal action, $\sup_{a\in\mathcal{A}}$), some approximation might be needed too, depending on the action space and its representation, and on the value function V. When this is the case, the approximation is encapsulated within the L Bellman operator (see Algorithm 3).

Regarding the former issue, i.e. computing $V_t(s)$ when state space $\mathcal{S}$ is large or continuous, standard supervised machine learning algorithms, more precisely regression algorithms are used to compute it backwards. Let us denote A a regression algorithm; A takes as input the Bellman image of the value function $L_{t+1}V_{t+1}$, and outputs the value function V_t. Formally, both the input and output domains of A are $\mathbb{R}^{\mathcal{S}}$ (functions from $\mathcal{S}$ to $\mathbb{R}$).

Several aspects must be considered in this regression stage. First of all, it makes sense to use active learning; learner A_t might usefully choose its training examples, see section 3.5. Secondly, it makes sense to consider different learning algorithms (or different parametrization) at every time step, for the sake of computational efficiency and convergence studies.

Although algorithm 3 looks simpler than algorithm 2, the complex issues are hidden in

Algorithm 3 Approximate Dynamic Programming

Input: a MDP, finite horizon T.
Input: a sequence of learners $A_t, t \in [[0, T-1]]$, $A_t : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$.
Initialize $\forall s \in \mathcal{S},\ V_T(s) = 0$
for $t = T-1$ **backwards to** 0 **do**
  $V_t = A_t L_{t+1} V_{t+1}$
end for
Output: $\forall t,\ V_t$.
Output: $\forall t, \forall s\ \pi_t(s) = \arg\max_{a \in \mathcal{A}} \left(r_t(s,a) + \int_{s' \in \mathcal{S}} p_t(s'|s,a) V_{t+1}(s') ds' \right)$.

the A_t learners[3]. Note that the value function V and policy π are no longer guaranteed to be optimal. The convergence of the process is not at stake, since there is a finite number of iterations.

Let us define more formally the notion of convergence and optimality with respect to V and π. Let n denote the number of examples used to learn the value function; let A_t^n, V_t^n and π_t^n respectively denote the *approximation operator*, the approximated value function and the corresponding policy. Let us consider the series V_t^n as n goes to infinity. Let us assume that A_t^n enforces some learning error $\varepsilon(n)$ with respect to the L_∞ norm; let us further assume that we can perfectly compute the operator L (the expectation and the optimization step $\sup_{a \in \mathcal{A}}$). Then the value function V_0^n (iteration n and time step 0) has an L_∞ error bounded by $T\varepsilon(n)$.
Therefore, if $\varepsilon(n)$ goes to 0 as n goes to infinity, the approximate value function converges toward the true one wrt L_∞ norm, which implies that policy π_t^n will also converge toward the optimal one.
The noise-free assumption regarding the expectation and optimization can be relaxed; if we instead bound the model and optimization errors (the respective bounds can easily be introduced through e.g. Monte-Carlo expectation approximation, and random optimizers), then V_t^n and π_t^n still converge toward the optimal ones with high probability.

The main difficulty raised by the above formalization is that most classical regression procedures and bounds refer to some L_p norm $p \geq 1$ rather than L_∞ norm. The use of an

[3] One can argue that everything is discrete in a computer, making the distinction between continuous and discrete framework only theoretical. However in practice we are never at the machine precision level, thus continuous notions (e.g. gradient), which has no sense in discrete, hold in a computer.

L_p norm requires one to define the underlying measure; due to the lack of prior knowledge, the Lebesgue measure will be considered in the following[4].

Unfortunately, the convergence of the learning error wrt L_p norm and Lebesgue measure does not imply the convergence of V_t^n and π_t^n toward the optimal value function and policy, as shown by the following counter-example.

> Let $\mathcal{S} = [0,1]$, $\mathcal{A} = [0,1]$ and assume that there are only two time steps (noted 0 and 1). The reward is always 0 except in the final time step, where the reward is 1 if and only if $|s_1 - \frac{1}{2}| \leq \frac{\varepsilon^p}{2}$ [5]. Consider a deterministic transition function, such that whatever the state s_0 (at time step 0), action a_0 leads to state $s_1 = a_0$ at time 1. The "true" value function at time step 1 $V_1(s_1)$ is 1 for $|s_1 - \frac{1}{2}| \leq \frac{\varepsilon^p}{2^p}$ and 0 elsewhere.
>
> Consider function $V_1' = -V_1$. By construction, $||V_1 - V_1'||_p = \varepsilon$. Still, the greedy policy based on V_1' is not ε-optimal; rather, it is the worst possible policy.
>
> This example illustrates that for any given ε, there exists a problem in which getting an approximation of the optimal value function at precision ε in L_p norm does not guarantee any upper bound on the policy error.

Additional assumptions, e.g. related to the "smoothness" of the MDP, are required for an efficient policy building using L_p -bounded learning of the value function (see [Mun07]).

Value iteration

The value iteration algorithm in infinite horizon is closely related to the stochastic dynamic programming described in the previous subsection. The goal is to solve the Bellman's equation $V = LV$, i.e. to find the fixed point of operator L. When considering the stationary case and a γ-discounted reward, the Bellman operator L is contractant; therefore the value iteration algorithm proceeds by iteratively applying L to some initial value function V. Algorithm 4 below considers finite state and action spaces. Following notations in algorithm

[4] Note that if the distribution of states, when the optimal policy is followed, was known, we could instead use this distribution to define L_p.

[5] Note that the reward could be made continuous and even C^∞ without changing the result. It would just make the counter-example technically more complex.

4, it terminates, ensuring that $||V_n - V_\gamma^*|| < \frac{2\gamma}{1-\gamma}\varepsilon$, with V_γ^* the value function of the optimal policy for discount γ.

When the state space is large or continuous, some approximation is needed. See e.g. [BD59, BT96] and [Mun07].

Algorithm 4 Value Iteration in finite state and action spaces

Input: a stationary MDP, with $\mathcal{S}$ and $\mathcal{A}$ finite.
Input: infinite horizon, γ-discounted, desired precision ε.
Initialize $\forall s \in \mathcal{S},\ V_0(s) = 0$
Initialize $n = 0$
repeat
  for all $s \in \mathcal{S}$ **do**
    $V_{n+1}(s) = L_\gamma V_n(s)$
  end for
  $n \leftarrow n+1$
until $||V_n - V_{n-1}|| < \varepsilon$
Output: V_n.
Output: $\forall s\ \pi(s) = \arg\max_{a\in\mathcal{A}} \left(r(s,a) + \gamma \int_{s'\in\mathcal{S}} p(s'|s,a)V_n(s')ds'\right)$.

Policy iteration

Another standard algorithm, the so-called "policy iteration" algorithm, is presented below in the stationary case, γ-discounted reward and finite $\mathcal{S}$ and $\mathcal{A}$ (algorithm 5).

This algorithm originates from the policy improvement result, stated as follows. Let π be a policy, and let π^+ be defined as follows[6]:

$$\forall s \in \mathcal{S},\ \pi^+(s) \in \arg\max_{a\in\mathcal{A}} \left(r(s,a) + \gamma \int_{s'\in\mathcal{S}} p(s'|s,a)V^\pi(s')ds'\right)$$

Then one has:

$$V_\gamma^{\pi^+} \geq V_\gamma^\pi$$

with $V_\gamma^{\pi^+} = V_\gamma^\pi \Leftrightarrow \pi = \pi^*$

It follows that each policy iteration results either in a strictly better policy or the same policy, and a finite sequence of policy improvements leads to the optimal policy (being

[6]Note that the ties in the argmax has to be broken in a systematic way in order to guarantee convergence.

reminded that a finite MDP admits a finite number of policies).

The limitation of the approach is that the estimation of V^{π} can be costly (first step in the repeat loop in algorithm 5, $\forall s \in \mathcal{S},\ V_n(s) = V_n^{\pi_n}(s)$). This step is handled through either Monte-Carlo approximation, or iterating operator I:

$$I(V)(s) = E_{\pi(s)}\left(r(s,\pi(s)) + \gamma \int_{s' \in \mathcal{S}} p_t(s'|s,\pi(s))V(s')ds'\right)$$

in order to estimate its fixed point. The fixed point can also be computed by solving a system of linear equations (operator I is linear): $\forall s,\ I(V)(s) = V(s)$.

Algorithm 5 Policy Iteration in finite state and action spaces

Input: a stationary MDP, with $\mathcal{S}$ and $\mathcal{A}$ finite.
Input: infinite horizon, γ-discounted.
Initialize π_0 an initial policy.
Initialize $n = 0$
repeat
  $\forall s \in \mathcal{S},\ V_n(s) = V_n^{\pi_n}(s)$
  for all $s \in \mathcal{S}$ **do**
    $\pi_{n+1}(s) = \arg\max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \int_{s' \in \mathcal{S}} p(s'|s,a)V_n(s')ds'\right)$.
  end for
  $n \leftarrow n+1$
until $\pi_n = \pi_{n-1}$
Output: $\pi_n = \pi^*$.

Next sections respectively present model-free and model-based methods for Reinforcement Learning.

1.2 Model-free methods

By construction, *model-free* methods do not build any explicit model of the environment, i.e. the transition $(s,a) \rightarrow s'$ and the reward functions are not explicitly represented.

1.2.1 $TD(0)$ algorithm

The so-called Temporal Difference (TD) algorithm first proposed by [Sut88] combines Monte-Carlo and Dynamic Programming approaches. This algorithm is widely used thanks to its efficiency and generality. It proceeds by iteratively updating the Q or V value function (see 1.1.3), adjusting the function at hand depending on the prediction error.

Algorithm 6 describes the simplest TD algorithm, referred to as $TD(0)$ and used to learn the value function of a given fixed policy (prediction). For the sake of simplicity, let us consider the stationary case, using a γ-discounted criterion (see section 1.1) with finite state and action spaces.

Let π be a policy. Among the parameters of the algorithm are series $\alpha_n(s)$ for each state s, where $\alpha_i(s)$ rules the size of each update step. It can be shown that function V converges towards V^π if $\forall s \in \mathcal{S},\ \sum_{i=0}^{\infty} \alpha_i(s) = \infty$ and $\sum_{i=0}^{\infty} \alpha_i(s)^2 < \infty$ and if every states are visited with a positive probability.

In practice, one most often chooses a constant series, $\forall s \in \mathcal{S},\ \forall n \in \mathbb{N},\ \alpha_n(s) = \alpha$, where α is a small constant; while the algorithm is empirically efficient is such cases, there is no longer any guarantee of convergence for the algorithm.

Algorithm 6 $TD(0)$ for prediction

Input: a stationary MDP, with $\mathcal{S}$ and $\mathcal{A}$ finite. A fixed policy π
Input: infinite horizon, γ-discounted.
Input: A policy π, a initial state $s_0 \in \mathcal{S}$.
Input: A sequence $\alpha_n(s)$, such that $\forall s \in \mathcal{S},\ \sum_n \alpha_n(s) = \infty$ and $\sum_n \alpha_n(s)^2 < \infty$.
Initialize $V_0(s)$ arbitrarily, e.g. $V_0(s) = 0$.
Initialize $s = s_0$, $n = 0$
while true **do**
  $a \leftarrow \pi(s)$.
  Take action a, observe reward r and next state s'.
  $V_{n+1}(s) \leftarrow V_n(s) + \alpha_n(s)\left[r - (V_n(s) - \gamma V_n(s'))\right]$.
  $s \leftarrow s'$, $n \leftarrow n+1$.
end while
Output: $V = V_\gamma^\pi$.

1.2.2 SARSA algorithm

the "SARSA" algorithm presented below is an adaptation of TD(0), aimed at learning the optimal policy π^* [SB98]. This algorithm, referred to as *On-Policy* learning, proceeds by iteratively updating the current policy π_n based on the current (state, action) pair. In contrast, in *Off-Policy* learning, updates are independent from the current policy; therefore the learned policy might differ from the one used in the action selection step.

The Sarsa algorithm is also closely related to the policy iteration algorithm (Algorithm 5, section 1.1.4): iteratively and alternatively, i) prediction function Q_n is updated; ii) the policy is updated and biased towards the optimal policy wrt Q_n. Formally, let us denote π_n a policy which acts consistently with Q_n. A widely used method is to define π_n as the ε-greedy policy ($0 \leq \varepsilon \leq 1$) wrt Q_n; with probability $1-\varepsilon$ the ε-greedy policy selects the best action according to Q_n ($= \arg\max_{a \in \mathcal{A}} Q_n(s,a)$) and with probability ε, it uniformly selects an action in $\mathcal{A}$. The (very simple) ε-greedy policy guarantees that every action is selected an infinite number of times with probability 1 irrespective of function Q_n.
Obviously, in most cases an ε-greedy policy cannot be optimal (non-optimal actions always have a probability > 0 to be selected). Under the condition that each state action pair is visited an infinite number of times and the policy converges in the limit to the greedy policy (e.g. setting π_n to the $\frac{1}{n}$-greedy policy on Q), convergence toward π^* can be guaranteed [SJLS00].

The Sarsa algorithm in the stationary, finite, γ-discounted MDP case is described below (Algorithm 7).

1.2.3 $TD(\lambda)$, $Sarsa(\lambda)$ algorithms and eligibility traces

Both above algorithms, $TD(0)$ and Sarsa, can be generalized into $TD(\lambda)$ and $Sarsa(\lambda)$ with $0 \leq \lambda \leq 1$. The λ parameter controls the length of time over which prediction errors are propagated. Each state is assigned an *eligibility trace*, indicating how often this state was visited recently; more precisely, the eligibility trace is the $\gamma\lambda$-discounted number of times the state was visited.

When $\lambda = 0$ only a one-step prediction error is considered: in the $TD(0)$ and Sarsa(0) algorithms described above only the current error ($r - (V(s) - \gamma V(s')$ or $r - (Q(s,a) -$

Algorithm 7 Sarsa: $TD(0)$ for control

Input: a stationary MDP, with $\mathcal{S}$ and $\mathcal{A}$ finite.
Input: infinite horizon, γ-discounted.
Input:
Let $(\alpha_j)_{j\in\mathbb{N}}$ denote a sequence such that $\sum_{i=0}^{\infty}\alpha_i = \infty$ and $\sum_{i=0}^{\infty}\alpha_i^2 < \infty$.
Define π_n a policy depending on Q_n
Initialize $Q_0(s,a)$ arbitrarily, e.g. $Q_0(s,a) = 0$.
Initialize $s = s_0$ (initial state in $\mathcal{S}$), $n = 0$
$a \leftarrow \pi_n(s)$
while true **do**
  Take action a, observe reward r and next state s'.
  $a' \leftarrow \pi_n(s')$
  $Q_{n+1}(s,a) \leftarrow Q_n(s,a) + \alpha_n\left[r - (Q_n(s,a) - \gamma Q_n(s',a'))\right]$.
  $s \leftarrow s'$, $a \leftarrow a'$, $n \leftarrow n+1$.
end while
Output: $Q = Q_\gamma^*$ if π_n explores each state-action pair an infinite number of times and converges toward the greedy policy on Q_n.

$\gamma Q(s',a'))$ respectively) is used to update the current state. When $\lambda > 0$ this prediction error is applied to all states in each iteration, in proportion to their eligibility trace. The prediction error is thus back-propagated to recently visited states. When $\lambda = 1$, the prediction error is back-propagated to all visited states, and the algorithms are then equivalent to Monte-Carlo prediction and Monte-Carlo control respectively.

The $TD(\lambda)$ algorithm learns the value function for a given policy (Algorithm 8, whereas the $Sarsa(\lambda)$ algorithm learns a policy (Algorithm 9). While these algorithms are more computationally expensive at each iteration (since each iteration requires a loop over all states[7]), they converge more quickly wrt the number of time steps. Because an additional experiment (involving the simulator or the real world) is more expensive than the update operator, the $TD(\lambda)$ and $Sarsa(\lambda)$ algorithms are efficient and popular.

Several alternative updates of eligibility traces have been considered in the literature; for example *replacing* traces can be used instead of the *accumulating* traces given in Algorithm 8. Denoting the current state by s_n, $e(s) \leftarrow \gamma\lambda e(s)$ if $s \neq s_n$ and $e(s) \leftarrow 1$ if $s = s_n$.

[7] A straightforward optimization is to only consider states with eligibility value $> \varepsilon$ in the loop.

Algorithm 8 $TD(\lambda)$ for prediction

Input: a stationary MDP, with $\mathcal{S}$ and $\mathcal{A}$ finite.
Input: infinite horizon, γ-discounted.
Input: A policy π, a initial state $s_0 \in \mathcal{S}$, a sequence α_n, $\lambda \in [0,1]$.
Initialize $V(s)$ arbitrarily, e.g. $\forall s,\ V(s) = 0$.
Initialize $\forall s,\ e(s) = 0$.
Initialize $s = s_0$, $n = 0$
while true **do**
  $a \leftarrow \pi(s)$.
  Take action a, observe reward r and next state s'.
  $\delta \leftarrow r - (V(s) - \gamma V(s'))$
  $e(s) \leftarrow e(s) + 1$.
  for all $s \in \mathcal{S}$ **do**
    $V(s) \leftarrow V(s) + \alpha_n \delta e(s)$.
    $e(s) \leftarrow \gamma\lambda e(s)$.
  end for
  $s \leftarrow s'$, $n \leftarrow n + 1$.
end while
Output: $V = V^{\pi}_{\gamma}$.

1.3 Model-based methods

Model-based methods explicitly build a model of the environment and internally use a planning algorithm to exploit this model. The contrast between model learning and value function learning (as described in previous sections) can be sketched as follows.
While value function learning can be viewed as indirect learning (the agent only observes the reward and the state/action pair), it easily derives a policy (e.g. the greedy policy based on Q or V).
Quite the contrary, model learning considers explicit information and can be handled through supervised learning: one gets state s' when taking action a in state s; therefore updating the transition $(s,a) \rightarrow s'$ is trivial. However, this model does not trivially derive a good policy.

Furthermore, model based methods allow for taking advantage of the possible factorizations in the environment. For instance, the notion of rotation of the robot can be grasped more easily in terms of model (the effect is independent of its position), than in terms of

Algorithm 9 *Sarsa*(λ): $TD(\lambda)$ for control

Input: a stationary MDP, with $\mathcal{S}$ and $\mathcal{A}$ finite.
Input: infinite horizon, γ-discounted.
Input: $\lambda \in [0,1]$.
Input:
Let $(\alpha_j)_{j\in\mathbb{N}}$ denote a sequence such as $\sum_{i=0}^{\infty}\alpha_i = \infty$ and $\sum_{i=0}^{\infty}\alpha_i^2 < \infty$.
Define π_n a policy depending on Q_n
Initialize $Q_0(s,a)$ arbitrarily.
Initialize $\forall s,a,\ e_0(s,a) = 0$.
Initialize $s = s_0$ (initial state in $\mathcal{S}$), $n = 0$
$a \leftarrow \pi_0(s_0)$
while true **do**
  Take action a, observe reward r and next state s'.
  $a' \leftarrow \pi_n(s')$
  $\delta \leftarrow r - (Q_n(s,a) - \gamma Q_n(s',a'))$
  $e(s,a) \leftarrow e(s,a) + 1$
  for all $s \in \mathcal{S}$, $a \in \mathcal{A}$ **do**
    $Q_{n+1}(s,a) \leftarrow Q_n(s,a) + \alpha_n \delta e(s)$.
    $e(s,a) \leftarrow \gamma\lambda e(s,a)$.
  end for
  $s \leftarrow s'$, $a \leftarrow a'$, $n \leftarrow n+1$.
end while
Output: $Q = Q_\gamma^*$ if π_n explores each state-action pair an infinite number of times and converges toward the greedy policy on Q_n.

value function (the value of the rotation indeed depends on the current position).

In summary, value functions are difficult to learn and easy to use, while models are easy to learn and difficult to use.

Some classical model-learning algorithms are presented here, which inspired several recent algorithms.

1.3.1 DYNA Architecture

The Dyna architecture, more a family of learning methods than an algorithm, originates from Sutton's *Dyna-PI* algorithm [Sut90]. This approach uses the policy iteration algorithm (section 1.1.4) to learn the policy from the model.

The Dyna principle is to simultaneously build a model of the transitions and a model of the expected rewards (using a value function, see previous subsections). At each iteration, i) the current experience is used to update the model; ii) the current policy gives the next action to take; iii) a incremental planner improves the current policy using the current model.

The so-called Prioritized Sweeping algorithm [AWM93] helps improve the model update efficiently, using a priority queue and sorting the states by (decreasing) prediction error.

1.3.2 E^3 algorithm

The *Explicit Explore and Exploit* algorithm (E^3) [MK02] also builds an explicit model of the environment. Here the exploration/exploitation dilemma is explicitly handled by maintaining a list of *known* transitions; other transitions are referred to as *unknown* transitions. Known transitions are the transitions which have been taken more than m times, where m is determined from the desired confidence interval $\pm\varepsilon$ and risk tolerance δ after Chernoff bound.
If the transition has been taken more than m times, then the probability of the estimate to be wrong by more than ε is smaller than δ. A *known* state is a state in which every transition is known.

The exploration policy aims at the uniform selection of all actions in any *unknown* state, by selecting the action which has been less often selected in this state (*balanced wandering*).

The E^3 algorithm comes with a formal proof of Probably Approximately Correct (PAC) non asymptotic optimality; more precisely the proof concerns the quality of the policy learned after a finite number n of experiences, while its quality refers to its behavior wrt infinite horizon (with no learning after time n).

Let us consider the γ discounted case[8].

Let M denote the real MDP, R_{max} an upper bound on the rewards, T the "horizon"[9] and opt the optimal return achieved by any policy on M within T time steps[10]. Let S be the current set of known states. The *known-state MDP*, noted M_S, is defined as follows: M_S includes all known states in M plus a single additional absorbing state s_0; it includes all transitions between known states in M; furthermore, every transition from some known state s to some unknown state s' in M is represented in M_S as a transition from s to s_0.

Informally, a policy is build on an approximation of M_S, switching between exploitation when the expected reward in the current approximation is almost optimal, and exploration when it worthes discovering new states.

The approximation of the transition probabilities and rewards on S leads to an approximation $\hat{M}_S$ of M_S. This approximation is used to build an efficient policy on $\hat{M}_S$, and it is shown that this policy is also approximately efficient on M_S as follows. Let M'_S be defined as a copy of M_S except for the reward function, set to R_{max} for s_0 and to 0 for all other states. Define $\hat{M}'_S$ as the current approximation of the optimal policy on M'_S (i.e. the policy leading to s_0, i.e. getting out of S, as quickly as possible).

Then, at each iteration:

- When the current state s is unknown ($s \notin S$) apply balanced wandering and update S;

[8] While [MK02] dealt with the non-discounted case, the analysis of the discounted case was presented in a longer version of [MK02].

[9] The horizon is defined as $\frac{1}{1-\gamma}$. Formally, [MK02] defines T as the **ε-return mixing time** i.e. the smallest T such as the per time step performance of the policy taken for all $T' \geq T$ is ε-close to the asymptotic one. While [MK02] also give some ways to remove the dependency of the algorithm in T, T is kept in the following for the sake of simplicity.

[10] The assumption of knowing opt can be removed, at the price of making more technically complex the description of the algorithm.

- when the current state s is known, a choice between exploitation and exploration is taken:

  - (Attempted Exploitation): compute off-line the optimal policy $\hat{\pi}$ for $\hat{M}_S$, and its reward at time step T in $\hat{M}_S$. If this reward is greater than $opt - \varepsilon/2$, then apply $\hat{\pi}$ for the next T steps. Else, choose the exploration:

  - (Attempted Exploration): compute off-line the optimal policy $\hat{\pi}'$ for $\hat{M'}_S$ and apply it for T steps. This policy will make the agent leave S with probability at least $\varepsilon/(2R_{max})$.

1.3.3 R_{max} algorithm

In the same spirit as E^3, the R_{max} algorithm introduced by Brafman [BT01] similarly maintains the list of known states; the main difference compared to E^3 is that R_{max} does not explicitly handle the Exploration vs Exploitation (EvE) tradeoff.

Formally, the MDP model M_S is built as in E^3, preserving all known states and transitions between them and adding an absorbing state. The underlying idea is simple: each unknown state gets an expected reward value equals to R_{max}, an upper bound on the rewards of this environment and is summarized absorbing state. Exploration is ensured as unknown states offer a high reward. The optimal policy thus simply decides between staying in the know states or going as quickly as possible to some unknown state. This tradeoff depends on the horizon time $T = \frac{1}{1-\gamma}$: if the rewards of the known states are high and T comparatively small, the exploration is not interesting. In the contrary, the R_{max} reward of the absorbing state representing the unknown states favors exploration. In addition to being simpler, R_{max} efficiently handles the EvE tradeoff and can be analyzed in a Game Theory perspective. Actually, many algorithms facing the EvE tradeoff (e.g. multi-armed bandit [ACBF02] or tree-structured multi-armed bandit [KS06]) are based on the same principle, referred to as “Optimism in the face of uncertainty”.

1.4 Summary

This introductory chapter has presented the domain of Reinforcement Learning, defining its formal background, describing some of the major algorithms at the state of the art, and discussing the many critical issues involved in the theory and practice of RL. Three of these issues will be more particularly considered in this manuscript.

The first major issue is that of scalability. It has been emphasized all over this chapter that the case of large finite, or continuous, action and state spaces, requires specific approaches and strategies. In this context, it is advisable to take advantage of the problem structure, e.g. using Factored MDPs [CBG95, BH99], and representing the MDP as a Dynamic Bayesian Network. This approach will be specifically investigated in Chapter 2.

A second critical issue regards the interplay of learning and decision, and the impact of the learning approximation on the decision optimality. This issue defines another RL challenge; the question is whether the stress should rather be put on the approximation (e.g. using the best ML tools) or on the optimization (e.g. using efficient sampling) steps; the difficulty is clearly that these steps are not independent. This issue will specifically tackled in chapter 3, devoted to Stochastic Dynamic Programming in continuous state and action spaces, focussing on non-linear optimization, regression learning and sampling.

A third issue concerns the Exploration vs Exploitation tradeoff. While this issue is acknowledged a key one for both domains of Reinforcement Learning and Game Theory (GT), little overlap between both domains has been considered in the literature until recently [AO06]. The possible overlap will be investigated in Chapter 4, considering a major challenge of AI since the end 90s, namely the domain of Computer-Go. This domain defines a high dimensional discrete RL problem, which exemplifies many ML challenges. A new algorithm inspired from GT, namely UCT [KS06] has been used and extensively adapted to the game of Go. The resulting program, MoGo, is the world strongest go program at the time of writing opening wide and exciting research perspectives.

Chapter 2

Data representation: Models and Bayesian Networks

This chapter is devoted to the representation of the environment, a key aspect of dynamic optimization problems as the success of most applications depends on the selected representation.

Several formalisms are described (section 2.1), taking robotics and specifically robotic mapping as motivating examples [FM02b, FM02a, Thr02]. This problem is representative of Partially Observable Markov Decision Process (POMDP) (see section 2.1.2 and Figure 2.2), where the actual state of the environment can only be guessed from the observations. In this case, robotic mapping is referred to as SLAM (Simultaneous Localization And Mapping) [Thr02].

The chapter thereafter focuses on Bayesian Networks (BN) [KP00] and Dynamic Bayesian Networks [Mur02], which have emerged as a popular formalism for compactly representing and solving large scale Markov Decision Processes (MDPs). The main contributions of the presented work concern:

- The definition of a new learning criterion, distinguishing the parametric and non parametric learning aspects (learning the parameters and the structure of the BN).
- The robustness properties of this criterion (wrt the guessed/learned structure), proved theoretically and empirically.

- The proposal of several algorithms to ensure the tractability of the optimization.
- Non asymptotic risk bounds on the learning error.

2.1 State of the art

This section lists the standard formalisms enabling the probabilistic representation of the environment[1].

2.1.1 Hidden Markov Models (HMMs)

Let us first introduce Hidden Markov Models (HMMs), referring the interested reader to the excellent tutorial [Rab89] for a comprehensive presentation. HMMs are a special case of Dynamic Bayesian Networks (DBNs), again a special case of Bayesian Networks (BNs) which are the central topic of this chapter.

Still, HMMs per se are directly or indirectly used as a model of environment in robotics and other domains, and particularly so in robotic mapping [FM02b, FM02a, Thr02]. While specific extensions of HMMs based on their factorization [GJ97],[BH01, HB01] or on a hierarchical decomposition (see respectively [MTKW02] and [TRM01] for their application in robotics), we here focus on the standard formalism.

An HMM is a stochastic finite automaton, involving a finite number of states noted $1 \ldots K$, where each state is associated a probability distribution on the observations. At time step t, X_t is the current (hidden) state and O_t is the associated observation (a discrete symbol or a real-valued vector). The HMM is characterized from:

- The initial state distribution π, where $\pi(i)$ is the probability of being in state i at time $t = 1$,
- The transition matrix $T(i, j) = P(X_t = j | X_{t-1} = i)$ (see Fig. 2.1),
- The distribution of the observations conditionally to the state, or observation model, $P(O_t | X_t)$.

[1] In a RL context, the reward function can be treated as an observation on these formalisms.

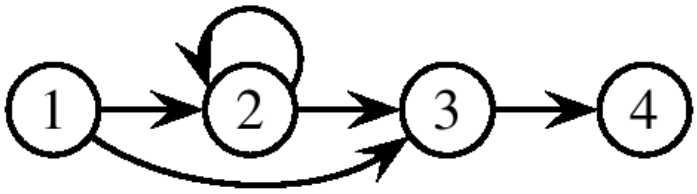

Figure 2.1: Graphical representation of a simple HMM (from [Mur02]), not representing the observation model. The four circles respectively represent the four states of the HMM. The *structure* of the HMM is given by the five arrows, depicting the (non zero) transition probabilities between nodes. The arrow from and to state 2 indicates the non zero probability of staying in state 2.

As the state space is discrete and finite, π represents a multinomial distribution. T is a distribution matrix (each row sums to one) and represents a conditional multinomial distribution. The set of pairs (i, j) such that $T(i, j) > 0$ is referred to as the *structure*[2] of the HMM.

In the case where observations are discrete symbols, the observation model is usually represented as a matrix $B(i,k) = P(O_t = k|X_t = i)$. When observations are real-valued vectors ($O_t \in \mathbb{R}^L$), the observation model is usually represented through Gaussian distributions, with mean μ and covariance matrix Σ ($\Sigma \in \mathbb{R}^{L\times L}$) depending on the state. Formally

$$P(O_t = o|X_t = i) = \mathcal{N}(\mu_i, \Sigma_i)(o) = \frac{1}{\sqrt{(2\pi)^L|\Sigma|}} exp\left(-\frac{1}{2}(o-\mu)'\Sigma^{-1}(o-\mu)\right)$$

2.1.2 Partially Observable Markov Decision Processes (POMDP)

A Partially Observable Markov Decision Processes (POMDP) can be seen as an extension of HMM, where the transition between states depends on the action A_{t-1} performed at time $t-1$; the decision making process is concerned with the selection of the action among a (usually) finite set of actions.

Accordingly, the transition matrix now defines the probability distribution for the state X_t conditioned by state X_{t-1} and action A_{t-1}, $T(i, j, k) = P(X_t = j|X_{t-1} = i, A_{t-1} = k)$.

[2]It must be noted that the word *structure* has different meanings in the context of a HMM or a Bayesian Network; see 2.1.5

The observation model, as in the HMMs, depends only on the current state X_t and can be written: $B(i,k) = P(O_t = k|X_t = i)$.

Fig. 2.2 depicts a POMDP (extracted from [TMK04]).

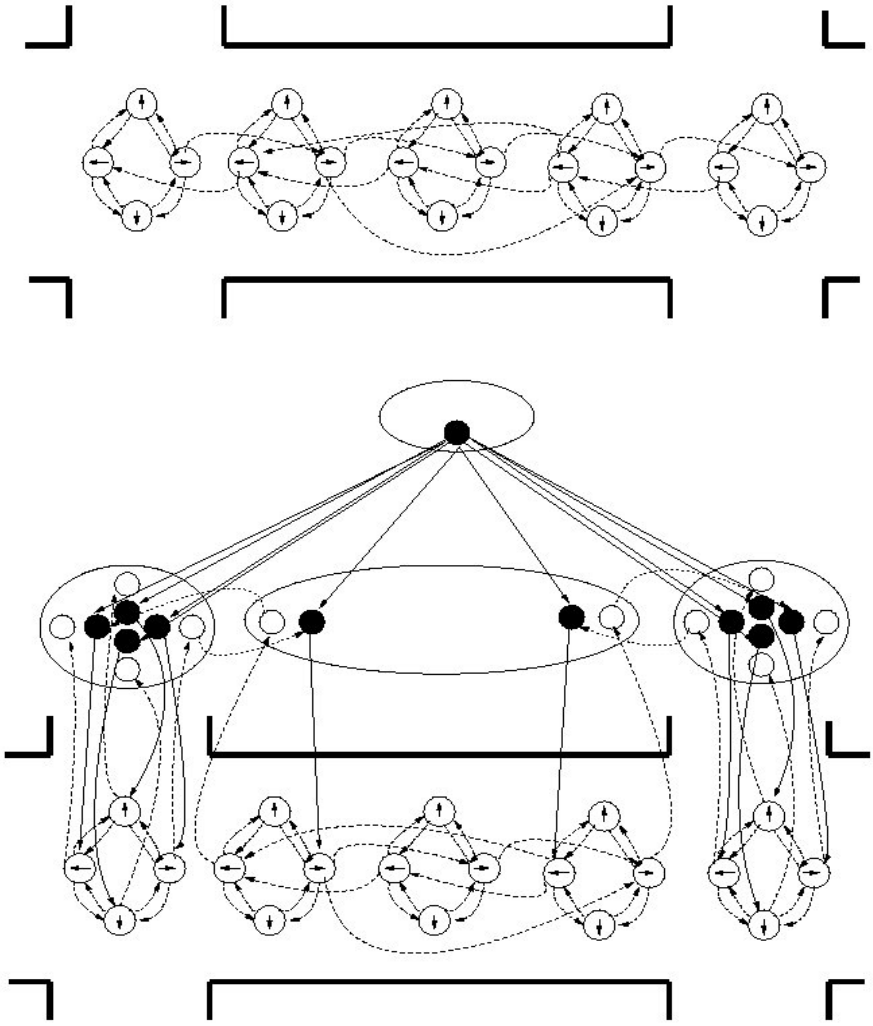

Figure 2.2: A POMDP in a robotic mapping application (from [TMK04], omitting the observations). The environment is a corridor with two crossings, alternatively represented as a flat POMDP (top) or a hierarchical POMDP (bottom). Each state, depicted as a circle, corresponds to a particular location and orientation of the robot (orientation given by the arrow inside the circle). Dashed lines indicate the transitions between states depending on the actions (rotation or translation) of the robot. White (resp. black) circles correspond to the entry (resp. exit) states.

2.1.3 Predictive State Representations (PSRs)

Predictive state representations (PSRs) [MLL01] are also concerned with the representation of discrete-time dynamic systems. The main difference between PSRs and HMMs is that the states refer to the past trajectory of the system in HMM, whereas they are defined in terms of what comes next (the next possible observations) in PSR and in Observable Operator Models (OOMs [Jae00]) alike. Specifically, the state of the system in PSRs is defined as a set of observable outcome predictions.

In other words, the PSR states are concerned with what may happen in the future, whereas HMM states characterize what happened in the past.

Following the definitions in [MLL01], the dynamic system at hand involves a discrete set of actions noted $\mathcal{A}$, where the observations range in a discrete set O. A *test* is a sequence of actions-observations, e.g. $a_1o_1a_2o_2...a_no_n$; which does not necessarily start at time step $t = 1$. When it starts at time step 1, a test is referred to as *history*. The *environment* is defined as a probability distribution over the set of all possible tests. The probability of a test is the probability of observing the $o_1,...,o_n$ (in that order) given that the actions $a_1,...,a_n$ are taken (in that order).

Given a set Q of tests ($Q = \{\tau_1,...,\tau_q\}$) the *prediction vector* is the function associating to a history h, the probability of each test conditioned by h (vector $p(h) = [P(\tau_1|h), P(\tau_2|h), ..., P(\tau_n|h)]$). This prediction vector is called a **predictive state representation (PSR)** if and only if it forms a sufficient statistic for the environment, i.e., if and only if for any test τ there exists a *projection function* $f_\tau : [0,1]^q \mapsto [0,1]$ such as for any test τ and history h

$$P(\tau|h) = f_\tau(p(h))$$

A *linear PSR* is a PSR for which there exists a *projection vector* m_τ for every test τ, such that for all histories h:

$$P(\tau|h) = f_\tau(p(h)) = p(h)m_\tau^T$$

The expressivity of PSR is higher than that of POMDP, as stated by the following theorem [MLL01]:

Theorem 2.1.1 (*POMDP* $\subset$ *PSR* (**[Jae00, MLL01]**)) *For any environment that can be represented by a finite POMDP model, there exists a linear PSR with number of tests no larger than the number of states in the minimal POMDP model.*

The above theorem states that linear PSRs can describe the same environments as POMDPs do, with no more states; furthermore, in some cases (e.g. in the *factored MDPs* [SC98]), the number of states of the PSR can be exponentially smaller than that of the equivalent POMDP [MLL01].

More generally, PSRs are proved to be strictly more general than POMDPs and n^{th}-order Markov models [SJR04]. Due to the relative novelty of these models, there are open problems regarding the learning of compact PSRs and their use for action selection.

PSRs learning has been pioneered by [SLJ$^+$03], using a gradient-based method for optimizing the parameters of a linear PSR; the main limitation of this work is that it assumes the tests to be given. The *discovery* problem (finding the tests of the PSR) was first addressed by [JS04], considering linear PSRs; the approach relies on the existence of a "reset" operator which can put the dynamical system in a fixed state.

More recently [BMJ$^+$06] succeeded in learning a PSR using non exploration policies, showing that an accurate model could be built with a significantly reduced amount of data.

Few works have been exploring the *planning* problem using PSRs, the first one being [IP03] using a policy iteration algorithm. [MRJL04] presents an iterative pruning algorithm and a Q-learning algorithm adapted to PSRs. As the state space of the PSR is continuous[3], the algorithm requires function approximation. The function approximators used were CMACs [Alb71], a grid-based method that uses r overlapping grids, each spanning the entire space of prediction.

2.1.4 Markov Random Fields

Markov Random Fields (MRFs) represent relationships between random variables using graphical models (the relationship wrt Bayesian Networks will be discussed in next section).

[3] Even with the world state space is discrete, a state of a PSR is a probability distribution, so is continuous.

Let $\mathcal{A} = \{A_1, ..., A_n\}$ denote a set of n random variables[4], with S its power set.
A MRF is defined from:

- A *neighborhood system*, $\mathcal{N} = \{N(A_i), i = 1, ..., n\}$, associating to each variable A_i in $\mathcal{A}$ its neighborhood $N(A_i) \subset \mathcal{A}$, with $A_i \in N(A_j)$ iff $A_j \in N(A_i)$;
- A *potential function* f, mapping each subset of variables to $\mathbb{R}$ ($f : S \mapsto \mathbb{R}$).

The neighborhood system is commonly represented through a graph $G = (\mathcal{A}, \mathcal{E})$, where $\{A_i, A_j\}$ is an edge in $\mathcal{E}$ iff $A_i \in N(A_j)$ (equivalently $A_j \in N(A_i)$).

The above defines a MRF iff the potential function f is positive, and every variable A_i only conditionally depends on the variables in its neighborhood:

$$\forall s \in S, f(s) > 0 \text{ (positivity)} \tag{2.1}$$

and

$$\forall i = 1 \ldots n, P(A_i | \mathcal{A} \setminus A_i) = P(A_i | N(A_i)) \text{ (markovianity)} \tag{2.2}$$

The positivity is assumed for technical reasons and can usually be satisfied in practice. For example, when the positivity condition is satisfied, the joint probability of any random field is uniquely determined by its local conditional probabilities [Bes]. The markovianity is here to model the locality of interactions between variables.

Thanks to the positivity property, potential function f can be written as an exponential function, often using a Gibbs distribution (MRF is then referred to as a Gibbs Random Field). Formally,

$$\forall s \in S, f(s) = Z^{-1} e^{-\frac{1}{T} U(s)}$$

where

$$Z = \sum_{s \in S} e^{-\frac{1}{T} U(s)}$$

is the normalizing constant called the *partition function*, T is a constant called the *temperature* and $U(s)$ is the *energy function*.

[4]As usual when dealing with graphical models, we, by abuse of notation, use $\mathcal{A}$ to denote either the set of random variables or the set of nodes. Likewise, we use A_i to denote the i^{th} variable or the corresponding node in the graph.

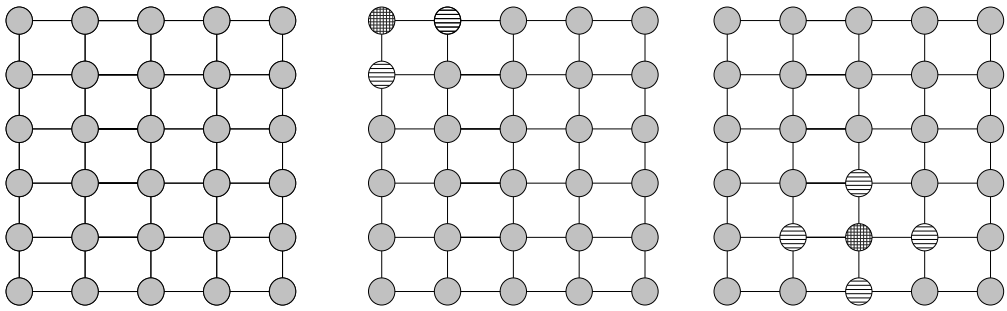

Figure 2.3: A grid-structured Markov Random Field (MRF). Each node represents a random variable, the neighborhood of which is given by the adjacent nodes. Left, center and right graphics depict a given node and the associated neighborhood.

Fig. 2.3 shows an example of MRF where the neighborhood of each node is the four nearest nodes in a 2D topology.

Some useful variants of Markov Random Fields are Conditional Random Fields [LMP01, Wal04, TD04], where each random variable can also be conditioned upon a set of global observations. The potential function f then also depends on the observations. This form of the Markov Random Field is more appropriate ([TD04]) for producing discriminative classifiers.

2.1.5 Bayesian Networks

Bayesian networks[5] (BNs), widely acknowledged for their ability of representing and reasoning on uncertain environments, are viewed as a marriage between the graph theory and the probability theory. The interested reader is referred to [Pea91, Pea00, NWL+04] for a comprehensive presentation of bayesian networks.

A BN is made of two components: the structure of the BN is a directed acyclic graph; and the conditional probabilities which can be represented by tabular or parametric models like Gaussian.

A very classical and simple example of BN is based on the following set of rules:

- If it is raining or sprinklers are on then the street is wet.
- If it is raining or sprinklers are on then the lawn is wet.
- If the lawn is wet then the soil is moist.
- If the soil is moist then the roses are OK.

These rules relate a set of random variables (it is raining, sprinklers are on, the street is wet, etc) and their relations can be represented by a graph (Fig. 2.4). Some probabilities can be put on these rules (e.g. expressing the probability for the roses to be OK if the soil is moist: $P(roses = ok|soil = moist) = 0.7)$).

Given the conditional probabilities on the set of rules (Fig. 2.4), the BN can be used to do reasoning, i.e. computing the probability of some variable knowing some others. For example, knowing that the roses are OK, what can be said about the state of the lawn? Answering such questions (e.g. computing $P(lawn = wet|roses = OK)$ and $P(lawn = dry|roses = OK)$) is referred to as *inference*.

Let R, S, L be the variables representing respectively $roses, sold, lawn$, it comes:

$$\begin{aligned} P(R,S,L) &= P(R|S,L)P(S|L)P(L) \\ &= P(R|S)P(S|L)P(L) \text{ for } R \text{ is independent of } L \text{ knowing } S \end{aligned} \quad (2.3)$$

[5] In spite of their name, Bayesian Networks are not "Bayesian" in the sense of the Bayesian decision theory [Rob94]; the word Bayesian here refers to the use of the "Bayes rule".

We have to work through soil first.

- $P(roses = OK|soil = moist) = 0.7$ and $P(roses = OK|soil = dry) = 0.2$;
- $P(soil = moist|lawn = wet) = 0.9$ then $P(soil = dry|lawn = wet) = 0.1$
- $P(soil = dry|lawn = dry) = 0.6$ then $P(soil = moist|lawn = dry) = 0.4$

We now compute $P(R,S,L) = P(R|S)P(S|L)P(L)$ (from equation 2.3):

- For R=ok, S=moist, L=wet, $0.7 \times 0.9 \times P(L = wet) = 0.63P(L = wet)$
- For R=ok, S=dry, L=wet, $0.2 \times 0.1 \times P(L = wet) = 0.02P(L = wet)$
- For R=ok, S=moist, L=dry, $0.7 \times 0.4 \times P(L = dry) = 0.28P(L = dry)$
- For R=ok, S=dry, L=dry, $0.2 \times 0.6 \times P(L = dry) = 0.12P(L = dry)$

Then, as

$$P(R = ok, L = wet) = P(R = ok, S = moist, L = wet) + P(R = ok, S = dry, L = wet)$$

it comes

$$P(R = ok, L = wet) = (0.63 + 0.02)P(L = wet)$$

.

We also have $P(R = ok, L = dry) = (0.28 + 0.12)P(L = dry)$.

Now $P(L|R = ok) = \frac{P(L,R)}{P(R=ok)} = \alpha P(L, R = ok)$ with $\alpha = \frac{1}{P(R=ok)}$ the normalizing constant.

Hence, $P(L = wet|R = ok) = 0.65\alpha$ and $P(L = dry|R = ok) = 0.3\alpha$. As

$$P(L = wet|R = ok) + P(L = dry|R = ok) = 1$$

(these are probabilities), it follows $\alpha = \frac{1}{0.95}$.

Hence $P(L = wet|R = ok) \approx 0.68$ and $P(L = dry|R = ok) \approx 0.32$.

In other words, given that roses are ok the lawn is likely to be wet.

This calculus exemplifies the use of BN for inference.

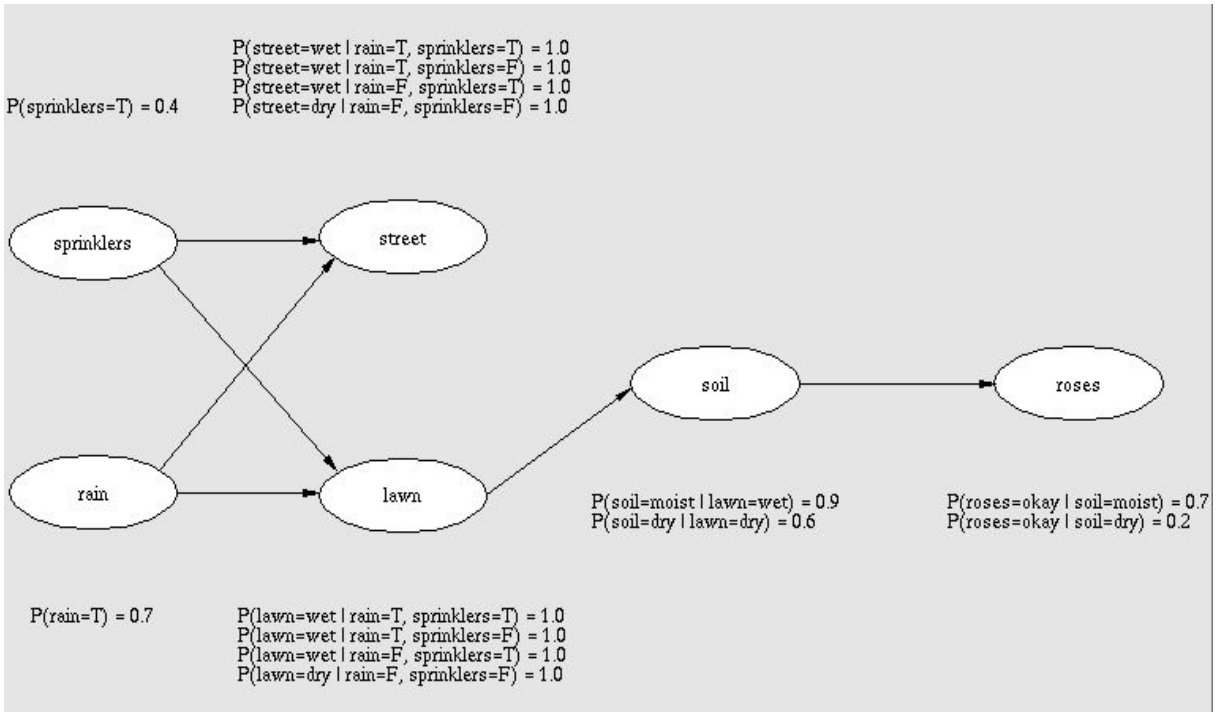

Figure 2.4: Classical example of a Bayesian Network

2.2 Scope of the problem

Identifying the structure and the parameters of bayesian networks can be based on expert knowledge, on data, or both. The rest of the chapter focuses on the second case, that is, learning a law of probability from a set of examples independently and identically distributed according to this law. Although many algorithms exist for learning bayesian networks from data, we address specific issues here.

Indeed, a bayesian model can be designed with different goals in mind: i) in order to evaluate probabilities (does a given event happen with probability 10^{-30} or 10^{-5}?); ii) in order to evaluate expectations (gains or losses of strategy $f(X)$).

In the first case, the point is to evaluate a risk; it comes naturally to use logarithms, amenable to maximum likelihood approaches.

In the second case, the point is to estimate $E_P(f)$, where P is the true probability distribution of random variable X. It might then be sufficient to use some approximation Q of P, since (Cauchy-Schwartz inequality)

$$|E_P(f) - E_Q(f)| \leq ||P - Q||_2 \times ||f||_2$$

In such cases, optimizing a monotonous function of L_2 norm ($||P - Q||_2$) seems to be the natural approach.

In the case where the current BN structure is not the right one, both approaches have very different robustness. Typically, maximum likelihood (frequency approach for probability estimation) leads to very unstable results; minimizing $||P - Q||_2$ offers a much better robustness (see 2.7.2 for details).

Therefore, the first contribution proposed in this chapter is a non-standard and tractable loss function for bayesian networks, with evidences of the relevance of this loss function for the L_2 criterion.

A second contribution is a complexity measure for bayesian networks models, involving a structural entropic term besides the standard number of parameters of the model. This new complexity measure is backed with theoretical results showing that:

- optimizing an ad hoc compromise between this complexity measure and the empirical L_2 error asymptotically leads to an optimal-in-size structure;
- the sample complexity of the network is directly related to the entropy term, and this term enables to distinguish among networks with same number of parameters (which get same scores after the usual measures);
- the complexity measure only depends on the class of distributions that is modeled by the network, i.e. we can work on Markov-equivalent structures.

The rest of this chapter is organized as follows: section 2.3 is a survey of BN learning and discusses our contribution compared to the state of the art; section 2.4 gives an overview of the presented results.

In section 2.5 we introduce formally the problem and the notations. Section 2.6 first recalls some classical results of learning theory and presents our result about evaluation of VC-dimensions and covering numbers. We then generalize our results to more general bayesian networks, with hidden variables, in section 2.6.5. Section 2.7 shows corollaries applied to structure learning, parameters learning, universal consistency, and others. Section 2.8 presents algorithmic details. Section 2.9 presents empirical results.

2.3 BN: State of the art

The problem of learning a bayesian network can be decomposed into a non-parametric and a parametric learning tasks ([NWL$^+$04]):

- Non-parametric learning is concerned with identifying the structure of the network, i.e. a graph;
- Given the structure, parametric learning is concerned with identifying the conditional probabilities among variables, i.e. the BN parameters.

The non-parametric learning task is clearly a more challenging problem than the parametric one.

2.3.1 Parametric learning

Given a Bayesian Network structure, i.e. a directed acyclic graph, the conditional probabilities for each variable can be learned from the data.

The classical approach for learning parameters is likelihood maximization. Classically decomposing the joint probability as a product, this leads to independently estimate each term of the product based on the dataset. This method asymptotically converges toward the true probability, *if* the considered structure is the true one. [Das97] studies [6] the sample complexity, i.e. how many examples are necessary to achieve a given accuracy.

The main other method, the bayesian method, aims instead at the most probable parameters given the data; using Bayes theorem, this amounts to biasing the most likely parameters with a prior. The prior most used in the literature is the Dirichlet distribution (see for example [Rob94]), because the Dirichlet prior is the conjugate prior[7] of the multinomial distribution.

2.3.2 Structure learning

While structure learning is NP-complete [Chi96], various methods have been proposed to efficiently learn the structure of a bayesian network under some restrictions.

Two types of approaches have been proposed for structure learning:

- The first one identifies dependencies (and independencies and conditional dependencies) among the random variables, and thereof deduces the structure of the graph;
- The second one relies on a score function, mapping every BN structure on the real value space; a "good" structure is found by optimizing the score function.

Since the space of all structures is super-exponential, heuristics must be defined in the latter, optimization based, approach (e.g. considering tree structures only, sorting the nodes, greedy search). The search can also be done in the space of Markov equivalent structures

[6] This work is particulary relevant as this approach also uses Covering Numbers, but for a different metric.

[7] A class of prior probability distribution is said to be conjugate to a class of likelihood functions if the posterior distributions are in the same family as the prior.

(in which structures that encode the same probability law are identified), which has better properties ([Chi02a]).

As our approach is based on a complexity measure which derives a score on BN structures, it belongs to the latter type of approaches. Notably, the proposed score is constant on Markov-equivalent structures.

While some works consider the influence of latent variables [GM98], the proposed complexity measure only takes into account the entropy of the dependency graph. The entropy is consistent with the number of latent variables; in particular, the bounds we provide hold in the latent variable case, although the score is the same as if all variables were observable. A direction for further improvement is based on [GM98], considering the statistical properties of networks with latent variables.

Learning dependencies

The task is to identify the independence (conditionally or not) among the variables; among the best known algorithms are IC and IC* [Pea00], PC [SGS93], and more recently BN-PC of Cheng et al. [CBL97a, CBL97b, CGK$^+$02].

Classically, the independence test is the χ^2 test. For hidden variables, the method is more complex as one must distinguish several types of dependencies; this issue goes beyond the scope of the presented work.

Score-based algorithms

Score-based approaches are generally based on Occam's razor principle, where the score of the structure measures its "complexity". Accordingly, the algorithm optimizes some tradeoff between the empirical error associated to the structure (quantified through the marginal likelihood or an approximation thereof [CH97]) and its complexity. This optimization can also be seen as a Bayesian model selection, selecting the structure S maximizing probability $P(S,D) = P(D|S)P(S)$ where D is the dataset; here, $P(D|S)$ measures the probability of the dataset given S, while $P(S)$ corresponds to the prior on S, biased toward simpler structures. Taking the log we get $logP(S,D) = logP(D|S) + logP(S)$, i.e. the sum of likelihood and the score of the structure $logP(S)$.

The best known scores for bayesian networks are listed below:

- AIC criteria [Aka70] or BIC [Sch78] use essentially $Dim(bn)$ to penalize the complexity of the bayesian network, where the "dimension" $Dim(bn)$ of the bayesian network simply is the number of parameters;
- The Minimum Description Length (MDL) principle [Ris78] uses the number of arcs and the number of bits used to code the parameters.
- The bayesian approach relies on a prior on the BN space. For example, the bayesian Dirichlet score [CH92a] assumes a Dirichlet prior on the parameters. Some variants exist (e.g. BDe [HGC94] or BDgamma [BK02]), using hyperparameters or priors on child/parent relations (given for example by an expert).

Given a score function, structural learning algorithms explore the structure space to find the structure with optimal score. One of the most common algorithms is the hill climbing one [Bun91], starting from an initial structure (given by an expert, another method, or the empty or full structure), computing the score of neighbor structures, replacing the current structure with its best neighbor and iterating until the stopping criterion is satisfied (resources exhausted or no more improvement).

The so-called $K2$ algorithm [CH92b] is a variant of the above greedy algorithm, parametrized from an *a priori* ordering over the variables, an upper bound on the number of parents, and a score function (besides the dataset). For each variable in the given order, $K2$ performs a hill climbing search, iteratively finding and adding the parent which maximizes the score, stopping when the upper bound on the number of parents is reached, or when the score does not improve anymore.

Other variants of the hill-climbing algorithm have been used, e.g. based on the variable neighborhood search [LMdC01]. Simulated annealing has also been used as optimization algorithm [JN06]; the difference with hill-climbing algorithms is that simulated annealing accepts no-better score structures with a probability which decreases along time, thereby enabling it to reach a local optimum with better quality.

Branch-and-bound algorithms [Suz96, Tia00] are more sophisticated algorithms, also extending the hill-climbing algorithms in order to better escape from local optima.

Population-based algorithms have also been applied to the problem of structure learning. All of them are stochastic algorithms ranging from genetic algorithms [LPY+96, MLD99], ant colonies optimization [dCFLGP02] to estimation of distributions algorithms [PLL04, RB03].

Instead of performing model selection (finding the structure with the highest score), the bayesian approach rather averages several structures, the main difficulty being to determine the number of structures to take into account. [FK03] uses a Monte-Carlo Markov Chain method to approximate the average efficiently in the space of permutations of the variables.

[FGW99] estimates confidence intervals on features (edges) of an induced model based on the bootstrap method.

The greedy algorithms defined in [Chi02b] and [CM02] consider the space of *Markov equivalent networks* instead of the space of structures. Two structures are equivalent if they can encode the same probability laws, with the same conditional independencies (more on this in section 2.6.4).

2.4 Overview of results

Usual parametric learning approaches (section 2.3.1) asymptotically lead to the optimal parameters if the BN structure is the right one. The advantage of the most usual frequentist approach is that it is very fast and only involves local variables (determining $P(B|A)$ only depends of the frequency of A,B-value combinations). A first contribution is to show however that this frequentist approach is unstable and not-optimal for some natural criterions if the structure does not match the decomposition of the joint law. By contrast, the proposed method, computationally harder and based on a global fitting of the parameters, is shown to be consistent (section 2.7.2).

The consistency result is based on risk bounds, showing that the probability of an error estimation larger than some ε is bounded by some δ depending on ε and the number of training examples. Equivalently, such bounds derive the *sample complexity* of the approach, i.e. the number of examples required to get an error lower than ε with probability at least $1-\delta$.

The case with hidden variables is addressed in section 2.6.5, considering both the

parametric and non-parametric learning tasks (section 2.7.3).

An algorithm providing universal consistency and asymptotic convergence towards the minimal L_2 error, asymptotically in the number of i.i.d examples is presented in section 2.7.4 (Thm 8). Moreover, we prove the convergence of the algorithm towards a minimal structure, in the sense of a user-defined complexity-measure (including classical optimality measures).

The comparison between our complexity measure and the usual ones gives insights into the main factors of structural complexity. The *covering numbers* of the BN associated to a given structure (Lemma 1) are directly related to the complexity of the structure. The bound established in Thm 7 depends on both the number of parameters R of the structure, and the "entropy" $H(r)$ of the structure, where $H(r) = -\sum_{k=1}^{a} \frac{r(k)}{R} ln(\frac{r(k)}{R})$ and $r(k)$ is the number of parameters of node k ($R = \sum_k r(k)$). It is shown empirically that (Fig. 2.10) $H(r)$ is correlated with the sample complexity R being fixed. Contrasting with the AIC, BIC and MDL measures (section 2.3.2), the proposed measure thus captures an aspect of complexity which goes beyond the number of parameters; the influence of the structure entropy is depicted on Fig. 2.5.

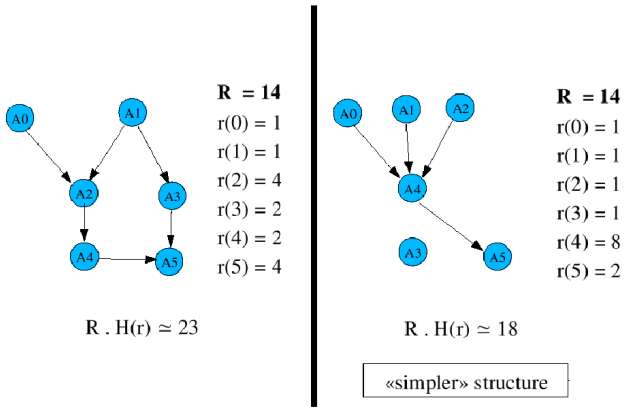

Figure 2.5: Scores of BN structures and influence of the entropy term. Both structures have the same number of parameters ($R = 14$). As they have different distributions of the parameters over the structure, they are associated different entropy terms; the right hand structure is considered "simpler" by our score.

Section 2.8 presents some algorithms handling the optimization of the proposed loss

criterion. These algorithms are based on the most standard quasi-Newton method [Bro70, Fle70, Gol70, Sha70], referred to as BFGS and relying on some non-trivial estimation of the loss function and its gradient.

Finally, the presented approach is empirically validated in section 2.9, showing both the statistical relevance of our approach and its tractability.

2.5 Problem definition and notations

For the sake of clarity and with no loss of generality, only binary random variables will be considered in the following. Let us first introduce the notations before presenting some preliminary results.

2.5.1 Notations

Let $A_1, \ldots A_p$ be p binary random variables. Let $\mathcal{U}$ denote a subset of $\{1,...,p\}$ and $A_{\mathcal{U}}$ be the product of variables A_i for i in $\mathcal{U}$.

We make the distinction between a Bayesian Network (*BN*), that is, a directed acyclic graph (DAG) and an instance of *BN*, defining the joint probability distribution over variables $A_1, \ldots A_p$ as the product of their conditional distributions as follows:

- a **bayesian network** noted *BN* is defined as a Directed Acyclic Graph $(\mathcal{A}, \mathcal{E})$ where $\mathcal{A} = \{A_1, \ldots, A_p\}$ is the set of nodes, corresponding to the random variables, and the set of edges $\mathcal{E}$ models the conditional dependencies between these variables. Formally, we denote $Pa(i) = \{j \in \{1,...,p\} \ s.t. \ (A_j, A_i) \in \mathcal{E}\}$ the set of variable indices j such that A_j is a parent of A_i in the graph. By definition of an acyclic graph, $i \notin Pa(i)$ and there is no sequence $i_1, i_2, \ldots, i_k$ such that $i_{j+1} \in Pa_{i_j}$ and $i_k = i_1$. With no loss of generality, it will be assumed that the indices are ordered after the DAG, that is, every j in $Pa(i)$ is such that $j < i$.

- an **instance of Bayesian Network** noted *ibn*, built on a $BN(\mathcal{A}, \mathcal{E})$, defines a joint

probability distribution on $\mathcal{A}$ as

$$ibn(\mathcal{A}) = \prod_j P(A_j | A_{Pa(j)})$$

For the sake of simplicity and by abuse of notations, an instance *ibn* built on some $BN(\mathcal{A}, \mathcal{E})$ will be said to "belong" to this *BN*, and will be noted $ibn \in BN$. To every state $v = (v_1, \ldots, v_p) \in \{0,1\}^p$, *ibn* associates the probability $Q(v) = P(A_1 = v_1, \ldots, A_p = v_p) \in [0,1]$. Equivalently, *ibn* can be defined as a 2^p-dimensional vector Q ($Q \in [0,1]^{2^p}$) of L_1 norm 1 ($||Q||_{L_1} = 1$). It will be noted Q_i, ($i \in \{1, \ldots, 2^p\}$), for the i^{th} component of the vector Q, or $Q(v)$, ($v \in \{0,1\}^p$), when Q is treated as a function.

The **number of parameters** of a bayesian network *BN*, noted $R = p(BN) = \sum_i 2^{|Pa(i)|}$, where notation $|U|$ stands for the cardinal of set U.

Let P denote a law of probability ; let a sample of examples be independent and identically distributed (iid), and let $\hat{P}$ denote the corresponding empirical law, that it, the set of Dirac masses located at the examples.

Let us further denote E (respectively $\hat{E}$) the expectation operator associated to P (resp. $\hat{P}$). Finally, let V be a multinomial random variable distributed according to P. Then let $\mathbf{1}_v = \mathbb{I}_{\{V=v\}}$ where $\mathbb{I}$ is the characteristic function[8].

For Q a vector in $[0,1]^{2^p}$ with $||Q||_{L_1} = 1$ (equivalent to a joint distribution over random variables $(A_1, \ldots, A_p)$, ie $Q(v) = Pr(A_1 = v_1, ..., A_p = v_p)$), we define the loss of Q noted $\mathcal{L}(Q)$ as follows:

$$\mathcal{L}(Q) = E(\sum_{v \in \{0,1\}^p} (Q(v) - \mathbf{1}(v))^2)$$

The empirical loss of Q noted $\hat{\mathcal{L}}(Q)$ is similarly defined as:

$$\hat{\mathcal{L}}(Q) = \hat{E}(\sum_{v \in \{0,1\}^p} (Q(v) - \mathbf{1}(v))^2)$$

Further, we associate to each *BN* the squared error $\mathcal{L}(BN)$ defined as the minimum, over all bayesian networks *ibn* built on structure *BN*, of $\mathcal{L}(ibn)$: $\mathcal{L}(BN) = min_{ibn \in BN} \mathcal{L}(ibn)$.

[8] $\mathbf{1}$ is equivalently a function from $\{0,1\}^p$ to $\{0,1\}$, with $\forall v \in \{0,1\}^p$, $P(\mathbf{1}(v) = 1) = P(A_1 = v_1, \ldots, A_p = v_p)$

2.5.2 Preliminary lemmas and propositions

The definitions above allow us to bound the loss with respect to the empirical loss, using the following Lemma.

Lemma: *Let us define*

$$N(Q) = \sum_{v \in \{0,1\}^p} (P(v) - Q(v))^2 \quad \hat{N}(Q) = \sum_{v \in \{0,1\}^p} (\hat{P}(v) - Q(v))^2$$

Then:

$$\mathcal{L}(Q) = N(Q) + 1 - \sum_{v \in \{0,1\}^p} P(v)^2$$

$$\hat{\mathcal{L}}(Q) = \hat{N}(Q) + 1 - \sum_{v \in \{0,1\}^p} \hat{P}(v)^2$$

Proof: We have:

$$\begin{aligned}
\mathcal{L}(Q) &= E\Big(\sum_{v \in \{0,1\}^p} (Q(v) - \mathbf{1}(v))^2\Big) \\
&= \sum_{v \in \{0,1\}^p} E((Q(v) - \mathbf{1}(v))^2) \\
&= \sum_{v \in \{0,1\}^p} \left[\mathrm{var}(Q(v) - \mathbf{1}(v)) + E(Q(v) - \mathbf{1}(v))^2\right] \\
&= \sum_{v \in \{0,1\}^p} \left[P(v)(1 - P(v)) + (Q(v) - P(v))^2\right] \\
&= 1 - \sum_{v \in \{0,1\}^p} P(v)^2 + N(Q)
\end{aligned} \tag{2.4}$$

The same applies for $\hat{\mathcal{L}}$. □

The following two propositions then hold.

Proposition A: *Let* δ *be a positive real value,* $0 < \delta < 1$ *and let* $\sup_\delta X$ *denote the* $1 - \delta$ *quantile of* X *(*$Pr(X < \sup_\delta X) > 1 - \delta$*).*

Let $x^* \in \operatorname{argmin} \mathcal{L} = \operatorname{argmin} N$[9]. *For all* $\hat{x} \in \operatorname{argmin} \hat{\mathcal{L}} = \operatorname{argmin} \hat{N}$*, with probability at least* $1-\delta$[10]*,*

$$\mathcal{L}(\hat{x}) \leq \mathcal{L}(x^*) + 2 \sup_{\delta} |\mathcal{L} - \hat{\mathcal{L}}|$$

Proof: The result follows from the three inequalities below; with probability at least $1-\delta$, the two first hold:

$$\begin{aligned} \hat{\mathcal{L}}(x^*) &\leq \mathcal{L}(x^*) + \sup_{\delta} |\mathcal{L} - \hat{\mathcal{L}}| \\ \mathcal{L}(\hat{x}) &\leq \hat{\mathcal{L}}(\hat{x}) + \sup_{\delta} |\mathcal{L} - \hat{\mathcal{L}}| \\ \hat{\mathcal{L}}(\hat{x}) &\leq \hat{\mathcal{L}}(x^*) \end{aligned}$$

□

From the Lemma and Proposition A, it is straightforward to see:

Proposition B: *With same notations as in Proposition A, it comes:*

$$N(\hat{x}) \leq N(x^*) + 2 \sup_{\delta} |\mathcal{L} - \hat{\mathcal{L}}|$$

As shown by these propositions, the empirical loss is closely related to the natural cost functions (N and $\hat{N}$), while being computable from the data.

2.6 Learning theory results

This section describes the theoretical results we obtained in the statistical learning framework, bounding the approximation error of a bayesian network using the two key notions of Vapnik-Cervonenkis dimension (VC-dim) and covering numbers. Let us first briefly introduce the formal background and notations in statistical learning.

[9] We recall that the argument of L is a probability distribution over $\{0,1\}^p$

[10] $\hat{L}$ being the empirical loss function, $\hat{L}$ is a random variable, depending on the random sample. Hence, $\hat{x} \in \operatorname{argmin} \hat{\mathcal{L}} = \operatorname{argmin} \hat{N}$ is also a random variable.

2.6.1 Introduction

The VC dimension ([VC71]), the most classical tool of learning theory, relates the learning accuracy to the size of the (hypothesis) search space. While VC-dim was first introduced in classification problems (the hypotheses map the instance space onto a discrete domain), it has been extended to regression problem and real-valued functions (see e.g. [Vap95b]). Taking inspiration from [WJB02, NSSS05], we use VC-dim to bound the loss of a Bayesian Network in section 2.6.2. Better bounds, based on the use of covering numbers (see e.g. [KT61]), are given in section 2.6.3.

The results we obtained follow the general Probably Approximately Correct (PAC) framework. The goal is to bound some quantity, usually the approximation error, by some small $\varepsilon > 0$ with some confidence $1 - \delta(\varepsilon)$, or some risk $\delta(\varepsilon)$, e.g.,

$$P(X > \varepsilon) < \delta(\varepsilon)$$

Equivalently, for some fixed risk $\delta > 0$ and some threshold $\varepsilon(\delta)$, the bound states that with risk δ (or with confidence $1 - \delta$), X is less than $\varepsilon(\delta)$.

Up to some fixed risk δ, the largest difference between empirical and expected loss (respectively $\hat{L}$ and L) within a function family $\mathcal{F}$ is often decreasing in $\frac{1}{\sqrt{n}}$, where n is the number of examples. Let us denote $F(\mathcal{F}, \delta)$ the smallest real value $\Delta > 0$ such that

$$P(\sup_{h \in \mathcal{F}} |\hat{L}(h) - L(h)| \geq \Delta/\sqrt{n}) \leq \delta$$

Note that, although $F(\mathcal{F}, \delta)$ depends upon n, the dependency in n can be omitted in many cases (i.e. the supremum on n is not a bad approximation). This notation is mainly to simplify the PAC bounds writing. Therefore we shall refer to $F(\mathcal{F}, \delta)$ in the following.

2.6.2 Bounds based on VC dimension

Let H_{BN} denote the set of bayesian networks *ibn* built on structure *BN*. With probability at least $1 - \delta$:

$$\sup_{ibn \in H_{BN}} |\hat{\mathcal{L}}(ibn) - \mathcal{L}(ibn)| \leq F(H_{BN}, \delta)/\sqrt{n}$$

The application $(a_1,\ldots,a_a) \mapsto \log P(A_1 = a_1,\ldots,A_a = a_a)$ is linear in the log of the parameters of the bayesian network. Since the combination with a monotonic function preserves the VC dimension, the VC dimension of H_{BN}, viewed as a family of applications mapping the state space $\{0,1\}^p$ onto $[0,1]$, is upper bounded by the number of parameters of *BN*. This shows:

Theorem C: *The VC dimension of the set H_{BN} of instanced bayesian networks is upper bounded by the number of parameters of* BN*, noted R. So thanks to classical results of learning theory, for a number of examples n greater than R,*

$$P(\exists \text{ibn} \in H_{BN} \;/\; |\hat{\mathcal{L}}(\text{ibn}) - \mathcal{L}(\text{ibn})| \geq \varepsilon) < 8(32e/\varepsilon)\log(128e/\varepsilon))^R \exp(-n\varepsilon^2/32)$$

Proof: These results are classical in learning theory. See e.g. [AB99, Th18.4 and 17.4].

2.6.3 Bound based on covering numbers

Covering numbers are as widely used as VC-dim in learning theory. Although inequalities of large deviations based on covering numbers are very loose and conservative, yet they are usually much tighter than those based on VC-dimension. Basically, the covering number $\mathcal{N}(H,\varepsilon)$ denotes the number of balls of radius ε needed to cover the search space H (Fig. 2.6).

Introduction

Let $\mathcal{N}(\mathcal{F},\varepsilon)$ denote the number of ε-balls (i.e. balls of radius ε) for a chosen distance d needed to cover function space $\mathcal{F}$. Given a set of such balls, the set of their centers is referred to as ε skeleton of $\mathcal{F}$ and noted $\mathcal{S}(\mathcal{F},\varepsilon)$.

Theorem C': *With same notations as for Thm C, and R denoting the number of parameters, the covering number of H_{BN} for the metric $d(\text{ibn}_1,\text{ibn}_2) = E(|\text{ibn}_1(\mathcal{A}) - \text{ibn}_2(\mathcal{A})|)$ is upper bounded by $e(R+1)(4e/\varepsilon)^R$.*

Proof: These results are classical in learning theory. See e.g. [AB99, Th18.4, p251]. □

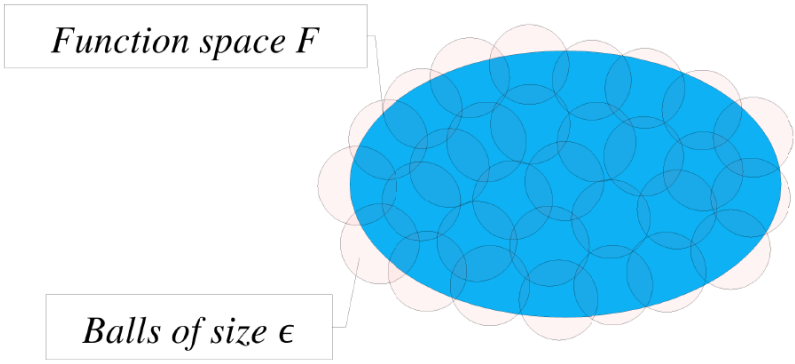

Figure 2.6: Covering function space F with balls of norm ε. The number of such balls needed to cover F, aka the covering number, reflects the complexity of F. It depends on the norm chosen and increases as ε decreases.

Then, with $\mathcal{L}$ and $\hat{\mathcal{L}}$ respectively a loss function and the corresponding empirical loss function (e.g. $\mathcal{L}$ and $\hat{\mathcal{L}}$ defined as previously), and assuming that $\mathcal{L}$ and $\hat{\mathcal{L}}$ map $\mathcal{F}$ onto $[0,2]$[11], the following results hold:

1. the risk, for any function f in $\mathcal{F}$, to have a deviation $|\mathcal{L}(f) - \hat{\mathcal{L}}(f)|$ more than 2ε, is bounded by $2\exp(-2n\varepsilon^2)$;

$$\forall f \in \mathcal{F}, Pr(|\hat{\mathcal{L}}(f) - \mathcal{L}(f)| > 2\varepsilon) \leq 2\exp(-2n\varepsilon^2)$$

2. The risk to have at least one of the centers of the balls having a deviation more than 2ε is upper bounded by $2\mathcal{N}(\mathcal{F},\varepsilon)\exp(-2n\varepsilon^2)$;

$$Pr(\exists f \in \mathcal{S}(\mathcal{F},\varepsilon)\ s.t.\ |\hat{\mathcal{L}}(f) - \mathcal{L}(f)| > 2\varepsilon) \leq 2\mathcal{N}(\mathcal{F},\varepsilon)\exp(-2n\varepsilon^2)$$

[11] Which holds true with $\mathcal{F} = H_{BN}$ and $\mathcal{L}$, $\hat{\mathcal{L}}$ defined as previously.

3. If $d(f,g) \leq \varepsilon \Rightarrow |\mathcal{L}(f) - \mathcal{L}(g)| \leq 2\varepsilon$ and $d(f,g) \leq \varepsilon \Rightarrow |\hat{\mathcal{L}}(f) - \hat{\mathcal{L}}(g)| \leq 2\varepsilon$[12], then the risk to have at least a function in $\mathcal{F}$ having a deviation more than 8ε is upper bounded by $2\mathcal{N}(\mathcal{F},\varepsilon)\exp(-2n\varepsilon^2)$.

$$\forall f \in \mathcal{F}, Pr(|\hat{\mathcal{L}}(f) - \mathcal{L}(f)| > 8\varepsilon) \leq 2\mathcal{N}(\mathcal{F},\varepsilon)\exp(-2n\varepsilon^2)$$

The above is straightforward by associating to every f its nearest neighbor g in the ε skeleton $\mathcal{S}(\mathcal{F},\varepsilon)$, and noting that:

$$|\hat{\mathcal{L}}(f) - \mathcal{L}(f)| \leq |\hat{\mathcal{L}}(f) - \hat{\mathcal{L}}(g)| + |\hat{\mathcal{L}}(g) - \mathcal{L}(g)| + |\mathcal{L}(g) - \mathcal{L}(f)| \leq 2\varepsilon + 4\varepsilon + 2\varepsilon = 8\varepsilon$$

The risk of having a function in $\mathcal{F}$ with deviation greater than ε is then upper bounded by $\delta = 2\mathcal{N}(\mathcal{F},\varepsilon/8)\exp(-2n(\varepsilon/8)^2)$.

Then it comes:

Proposition (maximal deviation for a given covering number):

$$\sqrt{n}F(\mathcal{F},\delta) \leq \inf\{\varepsilon | \log(2\mathcal{N}(\mathcal{F},\varepsilon/8)) - n\varepsilon^2/32 \leq \log\delta\}$$

Many variants of the above result are presented in the literature, see e.g. [Vid97a] and [AB99].

For a given structure *BN*, we shall now bound the covering number of $\mathcal{F} = H_{BN}$ with respect to the L_1 norm (noted $\mathcal{N}_1(H_{BN},\varepsilon)$), and use these results to bound the learning error.

Covering number of $\mathcal{F} = H_{BN}$

As noted in section 2.5.1, the bayesian structure *BN* is based on a directed acyclic graph (DAG) which induces an ordering on the variables (ranking every variable or node after its

[12] Which holds true with $\mathcal{F} = H_{BN}$, $\mathcal{L}$, $\hat{\mathcal{L}}$ defined as previously and the chosen distance d thanks to Lemma 2.

parent nodes).

Let $(E_k)_{k=1}^K$ denote a partition of nodes compatible with the DAG, such that:

- $\forall A_i \in E_k, A_j \in E_{k'}, (k \leq k') \Rightarrow (i \leq j)$
- There is no edge between two nodes of a same E_k:
  $\forall A_i \in E_k, A_j \in E_{k'}, (i \in Pa(j)) \Rightarrow (k < k')$

Such a partition $(E_k)_{k=1}^K$ always exists[13]. To each subset E_k of the partition we associate its depth k and the index l_k of the node with highest index in E_k. By convention, $E_0 = \emptyset$ and $l_0 = 0$. In the sequel, E_k is referred to as the k^{th} level of the bayesian network.

Bounding the covering number of H_{BN} is the subject of Thm 6 and 7, relying on Lemmas 1-5. Let us first introduce the following notations.

- n_k denotes the number of nodes of the bayesian network in level k (n_k= $|E_k|$,
- l_i is the number of nodes in levels $E_1, \ldots E_i$ ($l_i = \sum_{k=1}^i n_k$),
- F_k denotes the set of functions from $\{0,1\}^{l_k}$ onto $[0,1]^{l_k}$ defined by the first k layers of the bayesian network,
- T_k is the set of conditional probability tables associated to variables in E_k (the CPT defining $P(A_i|Pa(i))$ for $A_i \in E_k$),
- r_k is the number of rows of T_k ($r_k = \sum_{A_i \in E_k} 2^{|Pa(i)|}$).

Lemma 1:
The covering number of F_k wrt norm L_1 can be expressed wrt the covering number of F_{k-1} wrt norm L_1 and the covering number of T_k wrt norm L_∞:

$$\mathcal{N}_1(F_k, 2^{n_k}\varepsilon' + \varepsilon) \leq \mathcal{N}_1(F_{k-1}, \varepsilon)\mathcal{N}_\infty(T_k, \varepsilon')$$

[13]For instance E_k can be reduced to the k^{th} node in a topological order, with $K = p$. Many other partitions may exist, depending on the structure of the bayesian network.

Lemma 2:

$$|\mathcal{L}(Q) - \mathcal{L}(Q')| \leq 2||Q - Q'||_{L_1}$$

$$|\hat{\mathcal{L}}(Q) - \hat{\mathcal{L}}(Q')| \leq 2||Q - Q'||_{L_1}$$

Lemma 3:

$$\mathcal{N}_\infty([0,1]^h, \varepsilon) \leq \lceil \frac{1}{2\varepsilon} \rceil^h$$

Lemma 4:

$$\mathcal{N}_\infty(T_k, \varepsilon) \leq \lceil \frac{n_k}{2\varepsilon} \rceil^{r_k}$$

Lemma 5:

Let K be the number of levels E_k; then

$$\ln(\mathcal{N}_1(F_K, \varepsilon_K)) \leq \sum_{k=1}^{K} r_k \ln(\lceil \frac{n_k 2^{n_k - 1}}{\Delta_k} \rceil)$$

where $(\varepsilon_k)_{k=1}^K$ is a series of positive real values, $\varepsilon_{k-1} < \varepsilon_k$, with $\Delta_k = \varepsilon_k - \varepsilon_{k-1}$.

Theorem 6:

Let $R = \sum_{k=1}^K r_k$. Then

$$\ln(\mathcal{N}_1(F_K, \varepsilon)) \leq \sum_{k=1}^{K} r_k \ln(n_k 2^{n_k - 1} + \varepsilon) - \sum_{k=1}^{K} r_k \ln(\varepsilon r_k / R)$$

Theorem 7:

The above upper bound on the covering number is minimized for the partition $(E_k)_{k=1}^p$ made of singletons. The upper bound thus becomes:

$$\ln(\mathcal{N}_1(F_K, \varepsilon)) \leq \sum_{k=1}^{p} r_k \ln(1 + \varepsilon) - \sum_{k=1}^{p} r_k \ln(\varepsilon r_k / R) = R \ln((1 + \varepsilon)/\varepsilon) + RH(r)$$

where $H(r) = -\sum_{k=1}^p (r_k/R) \ln(r_k/R)$

Remark: Note that the above upper bound involves a term $log((1/\varepsilon)^R)$. The bound on $\mathcal{N}_1$ then exponentially increases with the number of parameters R as could have been

expected. The second term, the entropy of vector $(r_k)_{k=1}^p$, expresses that the covering number is at its minimum when the r_k distribution is as uniform as possible. Note also that asymptotically (ε close to 0), the upper bound mostly depends on the total number R of parameters, in the spirit of the BIC/AIC criteria [Sch78, Aka70].

These results will be combined to design a new penalization term, referred to as structural complexity penalization (section 2.6.3).

Proof of lemma 1:
Let $k \geq 1$ be fixed. Let $Pa(E_k)$ be the set of nodes that are parents of at least one node in E_k. Let $X \subset [0,1]^d$ be the set of d-dimensional vectors with L_1 norm 1, and $d = 2^{\sum_{i=1}^{k-1} |E_i|} = 2^{l_{k-1}}$. It is clear that every instanced bayesian network *ibn* built on *BN* up to level $k-1$ can be viewed as an element of X. Let Y similarly denote the set of d' dimensional vectors with L_1 norm 1, and $d' = 2^{l_k}$; Y likewise represents the set of all instanced bayesian networks built on *BN* up to level k.

Let y be a vector in Y, i.e. a probability distribution on the nodes in the first k levels. Let x denote the probability distribution on the nodes in the first $k-1$ levels marginalized from y. Let's note likewise $y' \in Y$ and the corresponding $x' \in X$. By construction, y is constructed from x through the conditional probability tables $P(A_i|Pa(i))$ for A_i in E_k, which can be viewed as an element t of T_k. $t' \in T_k$ corresponds to the construction of y' from x', giving $y = t(x)$ and $y' = t(x')$. It thus comes

$$\begin{aligned} ||y-y'||_{L_1} &= ||t(x)-t'(x')||_{L_1} \leq ||t(x)-t'(x)||_{L_1} + \underbrace{||t'(x)-t'(x')||_{L_1}}_{=||x-x'||_{L_1}} \\ &\leq \sum_{i=1}^{2^{n_k}} ||x||_{L_\infty} ||t(x)-t'(x)||_{L_\infty} + ||x-x'||_{L_1} \\ &\leq 2^{n_k} ||t-t'||_{L_\infty} + ||x-x'||_{L_1} \end{aligned}$$

It follows:

$$\mathcal{N}_1(F_k, 2^{n_k}\varepsilon' + \varepsilon) \leq \mathcal{N}_1(F_{k-1}, \varepsilon)\mathcal{N}_\infty(T_k, \varepsilon')$$

□

Proof of lemma 2:

$$|\mathcal{L}(Q)-\mathcal{L}(Q')| = |E\sum_i (Q_i-\mathbf{1}_i)^2 - \sum_i (Q'_i-\mathbf{1}_i)^2| \leq E|\sum_i (Q_i-\mathbf{1}_i)^2 - \sum_i (Q'_i-\mathbf{1}_i)^2|$$

$$\leq 2E|\sum_i |(Q_i-\mathbf{1}_i)-(Q'_i-\mathbf{1}_i)|| \leq 2E|\sum_i |Q_i-Q'_i|| \leq 2\sum_i |Q_i-Q'_i|$$

The same holds for $\hat{L}$. □

Proof of lemma 4: Consider some fixed k.

An element t of T_k can be viewed as the product of elements $t^{(i)}$ in $T_k^{(i)} = P(A_i|Pa(i))$, where $t^{(i)}$ defines the probability of $\wedge(A_l = a_l)$ for A_l ranging in $Pa(i)\cup A_i$. By construction $t^{(i)}$ belongs to $[0,1]^{|Pa(i)|}$ and its L_1 norm is 1. Let t and u be two elements in T_k. It comes

$$\begin{aligned} ||t-u||_{L_\infty} &= ||t^{(i_1)}\times\ldots\times t^{(in_k)} - u^{(i_1)}\times\ldots\times u^{(in_k)}||_{L_\infty} \\ &\leq ||t^{(i_1)}\times\ldots\times t^{(in_k)} - u^{(i_1)}\times t^{(i_2)}\times\ldots\times t^{(in_k)}||_{L_\infty} \\ &\quad +\ldots+||u^{(i_1)}\times\ldots\times t^{(in_k)} - u^{(i_1)}\times\ldots\times u^{(in_k)}||_{L_\infty} \\ &\leq ||t^{(i_1)}-u^{(i_1)}||_{L_\infty}+\ldots+||t^{(in_k)}-u^{(in_k)}||_{L_\infty} \end{aligned}$$

Therefore

$$||t-u||_{L_\infty} \leq n_k \times sup_{l_{k-1}<j\leq l_k}||t^{(j)}-u^{(j)}||_{L_\infty}$$

Finally, as t lives in $[0,1]^{r_k}$, after Lemma 3 it comes:

$$\mathcal{N}_\infty(T_k,\varepsilon) \leq \lceil\frac{1}{2\frac{\varepsilon}{n_k}}\rceil^{\Sigma_i |2^{Pa_{(i)}|}} = \lceil\frac{n_k}{2\varepsilon}\rceil^{r_k}$$

□

Proof of lemma 5:

After Lemma 4,

$$\mathcal{N}_\infty(T_k,\varepsilon) \leq \lceil\frac{n_k}{2\varepsilon}\rceil^{r_k}$$

Let K be the number of levels.

After Lemma 1, $\forall\varepsilon,\varepsilon' > 0$, $\forall 1 \leq k \leq K$:

$$\mathcal{N}_1(F_k, 2^{n_k}\varepsilon' + \varepsilon) \leq \mathcal{N}_1(F_{k-1}, \varepsilon)\mathcal{N}_\infty(T_k, \varepsilon')$$

Therefore, replacing $2^{n_k}\varepsilon' + \varepsilon$ by ε, and ε' by $\varepsilon'/2^{n_k}$:

$$\mathcal{N}_1(F_k, \varepsilon) \leq \mathcal{N}_1(F_{k-1}, \varepsilon - \varepsilon')\mathcal{N}_\infty(T_k, \frac{\varepsilon'}{2^{n_k}})$$

Further replacing ε' by $\varepsilon - \varepsilon'$:

$$\mathcal{N}_1(F_k, \varepsilon) \leq \mathcal{N}_1(F_{k-1}, \varepsilon')N_\infty(T_k, \frac{\varepsilon - \varepsilon'}{2^{n_k}})$$

Setting $\varepsilon = \varepsilon_k, \varepsilon' = \varepsilon_{k-1}$ gives, for all $\varepsilon = \varepsilon_K \geq 0$ (reminding that $r_k = 2^{n_k}$),

$$\begin{aligned} \ln(\mathcal{N}_1(F_K, \varepsilon)) &\leq \sum_{k=1}^K r_k \ln(N_\infty(T_k, \varepsilon_k - \varepsilon_{k-1})) \\ &\leq \sum_{k=1}^K r_k \ln(\frac{n_k 2^{n_{k-1}}}{\varepsilon_k - \varepsilon_{k-1}}) \\ &\leq \sum_{k=1}^K r_k \ln(\frac{n_k 2^{n_{k-1}}}{\Delta_k}) \end{aligned}$$

□

Proof of theorem 6:

Tranforming Lemma 5, it comes:

$$\ln(\mathcal{N}_1(F_K, \varepsilon)) \leq \sum_{k=1}^K r_k \ln(n_k 2^{n_k - 1} + \Delta_k) - \sum_{k=1}^K r_k \ln(\Delta_k)$$

Bounding Δ_k by ε,

$$\ln(\mathcal{N}_1(F_K, \varepsilon)) \leq \sum_{k=1}^K r_k \ln(n_k 2^{n_k - 1} + \varepsilon) - \sum_{k=1}^K r_k \ln(\Delta_k)$$

As we want to maximize term $\sum r_k \ln(\Delta_k)$ subject to constraint $\sum \Delta_k = \varepsilon$, the Kuhn-Tucker condition gives $\Delta_k = \frac{r_k}{\sum_k r_k}$ and finally:

$$\ln(\mathcal{N}_1(F_K, \varepsilon)) \leq \sum_{k=1}^K r_k \ln(n_k 2^{n_k - 1} + \varepsilon) - \sum_{k=1}^K r_k \ln(\varepsilon r_k / R)$$

□

Proof of theorem 7: Theorem 6 holds true conditionally to the fact that the partition is compatible with the DAG structure ($E_k \cap Pa(k) = \emptyset$). We show that the upper bound (left hand side) reaches its minimum when the partition is made of singletons.

Let s_i denote the number of parameters in the CPT of variable A_i. By definition, for all $1 \leq k \leq K$, $r_k = \sum_{A_i \in E_k} s_i$.

Let us denote $k(i)$ such as $E_{k(i)}$ includes variable A_i. After Theorem 6 it comes:

$$\begin{aligned} \ln(\mathcal{N}_1(F_K, \varepsilon)) &\leq \sum_{k=1}^{K} \left[\sum_{A_i \in E_k} s_i \ln \left(\frac{R(n_k 2^{n_{k-1}} + \varepsilon)}{\varepsilon \sum_{A_i \in E_k} s_i} \right) \right] \\ &\leq \sum_{i=1}^{p} s_i \ln \left(\frac{R(C(E_{k(i)}) + \varepsilon)}{\varepsilon \sum_{\{j \ / \ A_i \in E_{k(j)}\}} s_j} \right) \\ \textit{where} & \\ & C(E_k) = (|E_k|)\, 2^{|E_k|-1} \end{aligned}$$

Let us assume that some E_k is not a singleton (E_k includes m variables, $|E_k| = m$); with no loss of generality, $k = 1$. With no loss of generality, let $A_1, \ldots A_m$ denote the variables in E_1 and assume that variable A_1 reaches the minimal complexity in E_1 ($s_1 = argmin\{s_i, A_i \in E_1\}$).

Let us split E_1 into two subsets, where the first subset is made of the singleton $E_1' = \{A_1\}$ and $E_1'' = E_1 \setminus \{A_1\}$. Note that if the former partition was compatible with the DAG structure, the latter one is compatible too. The right hand side in Thm 6 respectively corresponding to E_1 and E_1', E_1'' are written below:

$$\begin{aligned} r.h.s.(E_1) &= \sum_{i=1}^{m} s_i ln \left(\frac{C(E_1)}{\sum_{i=1}^{m} s_i} \right) \\ &= s_1\, ln \left(\frac{C(E_1)}{\sum_{i=1}^{m} s_i} \right) + \sum_{i=2}^{m} s_i ln \left(\frac{C(E_1)}{\sum_{i=1}^{m} s_i} \right) \end{aligned}$$

$$\begin{aligned} r.h.s.(E_1') &= s_1 ln(\tfrac{1}{s_1}) \\ r.h.s.(E_1'') &= \sum_{i=2}^{m} s_i ln \left(\frac{C(E_1'')}{\sum_{i=2}^{m} s_i} \right) \end{aligned}$$

Computing the difference d between the former and latter r.h.s gives:

$$\begin{array}{lll} d & = & \sum_{i=2}^{m} s_i \ln\left(\frac{C(E_1)\sum_{i=2}^{m} s_i}{C(E_1'')\sum_{i=1}^{m} s_i}\right) + s_1 \ln\left(\frac{C(E_1)s_1}{\sum_{i=1}^{m} s_i}\right) \\ & = & A\ln\left(\frac{C(E_1)A}{C(E_1'')(A+B)}\right) + B\ln\left(\frac{C(E_1)B}{A+B}\right) \\ where & & A = \sum_{i=2}^{m} s_i \\ & & B = s_1 \end{array}$$

Therefore, with $\lambda = \frac{A}{A+B}$:

$$\frac{d}{A+B} = \lambda \ln(\lambda C(l)/C(l-1)) + (1-\lambda) ln(C(l)(1-\lambda))$$

The above expression reaches its minimum for $\lambda = \frac{C(E_1'')}{1+C(E_1'')}$. As the minimum is positive ($0 \leq \ln(\frac{C(E_1)}{1+C(E_1'')})$), it comes that the r.h.s. in Theorem 6 is lower for the splitted partition, which concludes the proof. □

Summary of the results

The above results provide an upper bound on the covering number for the set of bayesian networks built on a given structure BN, associated to a total number of parameters $R = \sum_{i=1}^{p} s_i$.

Theorem 7 states that for all $\varepsilon > 0$:

$$\ln(\mathcal{N}_1(F_K, \varepsilon)) \leq \sum_{i=1}^{p} s_i \ln(1+\varepsilon) - \sum_{i=1}^{p} s_i \ln(\varepsilon s_i/\mathcal{S}) \tag{2.5}$$

Lemma 2 states that the approximation error L and its empirical estimate $\hat{L}$ are Lipschitz; $d(f,g) \leq \varepsilon \Rightarrow |L(f) - L(g)| \leq 2\varepsilon$ and $d(f,g) \leq \varepsilon \Rightarrow |\hat{L}(f) - \hat{L}(g)| \leq 2\varepsilon$ hold true. It thus follows, after the results stated in subsection 2.6.3, that the risk of having a deviation between L and $\hat{L}$ greater than ε in function set $\mathcal{F}$ can be upper bounded as follows:

$$Pr(|L(f) - \hat{L}(f)| \geq \varepsilon) \leq \delta = 2\mathcal{N}_1(\mathcal{F}, \varepsilon/8)\exp(-2n(\varepsilon/8)^2)$$

Therefore,

$$F(\mathcal{F},\delta) \leq \sqrt{n}\inf\{\varepsilon | \log(2\mathcal{N}_{\mathbb{J}}(\mathcal{F},\varepsilon/8)) - n\varepsilon^2/32 \leq \log\delta\}$$

And finally:

$$P(\sup_{h\in BN} |\hat{L}(h) - L(h)| > \varepsilon) \leq 2\mathcal{N}_{\mathbb{J}}(BN,\varepsilon/8)\exp(-n\varepsilon^2/32) \tag{2.6}$$

Using equation (2.5), equation (2.6) can be solved in ε. Letting R denote the number of parameters in BN, H the corresponding entropy, n the number of examples involved in the computation of the empirical estimate and δ the desired confidence, it comes:

$$C = 2(H - \frac{1}{R}\log(\delta/2)) \qquad B = \frac{4n}{R}\exp(C)$$
$$A = -R \times W_{Lambert}(B) - 2\log(\delta/2) + 2RH \qquad \varepsilon = \frac{8}{\exp(-A/(2R))}$$

$$\varepsilon = 8\left(\frac{\delta}{2}\right)^{-\frac{1}{R}} \exp\left(-\frac{1}{2}W_{Lambert}\left(\frac{4n}{R}\frac{e^{2H}}{\frac{\delta}{2}^{\frac{2}{R}}}\right) - \frac{1}{R} + H\right)$$

Where $W_{Lambert}$ is the function such as $W_{Lambert}(x) \times e^{W_{Lambert}(x)} = x$.

It follows that the generalization error is upper bounded by the sum of the empirical error $\hat{L}$ and the complexity term $\varepsilon(R, H(r), n, \delta)$ as above. When the number R of parameters is large, the complexity term is dominated by $4\exp(H)$.

The difference, compared to standard structural risk minimization, goes as follows. Structural risk minimization classically proceeds by minimizing $\hat{L}(s) + R(m(s))$ where R is some regularization term, m is some complexity measure, s is the structure, $\hat{L}(s)$ is the minimal empirical error with structure s. Knowing the covering numbers of $\{s; m(s) \leq M\}$ as a function of M enables to derive the regularization criterion $R(.)$ such that minimizing $\hat{L}(s) + R(m(s))$ ensures universal consistency.

The proposed approach is based instead on the covering number associated to a fixed structure BN, as opposed to, the set of all structures with complexity upper bounded by some quantity M. This difference will be dealt with either (i) a priori choosing a sequence of embedded structures (see Corollary C3 below, 2.7.3); or (ii) spreading the risk δ among

structures with a given number of parameters. Considering $\varepsilon(R, H(r), n, \delta/N)$ where N is the number of structures with less than R parameters will lead to bound the deviation $L - \hat{L}$ uniformly on all structures with less that R parameters[14].

The next step is to show that the complexity term in equation (2.6) is constant over Markov-equivalent structures.

2.6.4 Score-equivalence: the score is the same for Markov-equivalent structures

Two bayesian networks are Markov-equivalent if they encode the same probability laws [Chi02a]; Therefore, it is important that the complexity term in equation (2.6) is fully compliant with the equivalence classes. Let $\mathcal{G}$ denote a DAG, and $Dim(\mathcal{G})$ the number of parameters in the bayesian network with structure $\mathcal{G}$.

The rest of this section shows that two Markov-equivalent graphs have same number of parameters R, same entropy $H(r)$ and same empirical approximation error $\hat{L}$; proofs use Lemma 1, Theorem 2 and Theorem 3 from [Chi95].

Reminder

For the sake of readability, let us remind that $\mathcal{G} = (U, E_{\mathcal{G}})$ an acyclic directed graph (DAG) i.e. a bayesian network. $Pa(x)_{\mathcal{G}}$ denotes the set of the parent nodes of node x in $\mathcal{G}$. We note $\mathcal{G} \approx G'$ if the bayesian networks based on $\mathcal{G}$ and $\mathcal{G}'$ are equivalent.

Definition (definition 2 in [Chi95]) *An edge $e = x \rightarrow y \in E_{\mathcal{G}}$ is covered in $\mathcal{G}$ if $Pa(y)_{\mathcal{G}} = Pa(x)_{\mathcal{G}} \cup \{x\}$.*

Lemma A (lemma 1 in [Chi95]) *Let $\mathcal{G}$ be a DAG containing the edge $x \rightarrow y$, and let $\mathcal{G}'$ be the directed graph identical to $\mathcal{G}$ except that the edge between x and y in $\mathcal{G}'$ is $y \rightarrow x$. Then $\mathcal{G}'$ is a DAG that is equivalent to $\mathcal{G}$ if and only if $x \rightarrow y$ is a covered edge in $\mathcal{G}$.*

Lemma B (theorem 2 in [Chi95]) *Let $\mathcal{G}$ and $\mathcal{G}'$ be any pair of DAGs such that $\mathcal{G} \approx G'$. There exists a sequence $\mathcal{G}, \mathcal{G}_1 = r_1(\mathcal{G}), \ldots, \mathcal{G}_N = r_N(\mathcal{G}_{N-1}) = \mathcal{G}'$, where r_i only reverses an edge in $\mathcal{G}_{i-1}$ and*

[14]Dividing confidence δ by the number N of such structures provides very conservative bounds; finer-grained approaches, unequally spreading the risk, are more appropriate [LB03].

- *The edge reversed by r_i is a covered edge in $\mathcal{G}_{i-1}$.*
- *Each $r_i(\mathcal{G})$ is a DAG and $r_i(\mathcal{G}) \approx \mathcal{G}'$.*
- *Every $\mathcal{G}_i$ is a DAG and is equivalent to $\mathcal{G}'$.*

Lemma C (theorem 3 in [Chi95]) *If $\mathcal{G} \approx G'$ then $Dim(\mathcal{G}) = Dim(\mathcal{G}')$.*

Results

Lemma C shows that two equivalent structures have same number of parameters.

In order to show that two equivalent structures $\mathcal{G}$ and $\mathcal{G}'$ have same entropy, after Lemma B it is enough to show that reversing an edge $x \to y$ which is covered in $\mathcal{G}$ does not modify the entropy. Denoting $r_{\mathcal{G}}(x)$ the number of parameters of node x in $\mathcal{G}$, it comes

- $r_{\mathcal{G}}(y) = 2r_{\mathcal{G}}(x)$.
- By symmetry, $y \to x$ is covered in $\mathcal{G}'$, therefore $Pa(x)_{\mathcal{G}'} = Pa(y)_{\mathcal{G}'} \cup \{y\}$, and $r_{\mathcal{G}'}(x) = 2r_{\mathcal{G}'}(y)$.
- Since the only difference between $\mathcal{G}$ and $\mathcal{G}'$ concerns edge $y \to x$:
  - the only changing terms in $H(r)$ is those corresponding to x and y.
  - and also $r_{\mathcal{G}}(y) = 2r_{\mathcal{G}'}(y)$
- Therefore $r_{\mathcal{G}'}(y) = r_{\mathcal{G}}(x)$ and $r_{\mathcal{G}'}(x) = r_{\mathcal{G}}(y)$

Which concludes the proof: $\mathcal{G}$ and $\mathcal{G}'$ have same entropy.

Finally, as two equivalent structures of bayesian networks encode the same space of probability distributions, their empirical loss is equal.

Therefore, all learning criteria only depending on R, $H(r)$ and $\hat{L}$ are *Markov-equivalent*.

2.6.5 Results with hidden variables

Let us consider the case where the probability law involves hidden variables: hidden variables are not considered (as they are not observed) when computing L or $\hat{L}$. It must be emphasized that learning a BN with hidden variables is significantly different from learning a BN only involving the subset of observed variables. Typically, a network with one hidden variable B and d observed variables $A_1 \dots A_d$ where each A_i depends on B, only involves $2d+1$ parameters; it would require many more parameters to be modelled as a BN only considering the A_is.

Let us represent a BN over p variables as a vector (with L_1 norm 1) of dimension 2^p (assuming, as in all this chapter, that all variables are binary). A BN involving $p-d$ hidden variables can be represented as a vector of dimension 2^d, where d is the number of observed variables, by marginalizing the probabilities over the hidden variables. It is clear that marginalization is 1-Lipschitz, that is: if x and y are two BN with p variables among which d are non-hidden, $\tilde{x}$ and $\tilde{y}$ are the laws derived from x and y and expressed as vectors of dimension 2^d, then $||\tilde{x}-\tilde{y}||_1 \leq ||x-y||_1$. For each instantiated network $ibn \in BN$ we associate $i\tilde{bn}$ the instantiated network after marginalization of all hidden variables. We note $\tilde{BN} = \{i\tilde{bn}/ibn \in BN\}$.

Therefore, the deviation between the true and the empirical error can be bounded as follows. **Proposition maximal deviation in a bayesian network with hidden variables:**

The risk to have a deviation at least ε *for a* $i\tilde{bn} \in \tilde{bn}$ *is upper bounded as follows:*

$$P(\sup_{i\tilde{bn}\in\tilde{\mathrm{BN}}} |\hat{L}(i\tilde{bn}) - L(\tilde{\mathrm{BN}})| > \varepsilon) \leq 2N1(\mathrm{BN}, \varepsilon/8)\exp(-n\varepsilon^2/32)$$

.

Remarks: Note that, despite the fact that $\tilde{ibn}$ involves only $d < p$ variables, the bound remains the same as when p variables are considered. The alternative, if the number of hidden variables is very large, would be to consider the covering number associated to vectors of dimension 2^d ($F(\{0,1\}^d, \delta)$) instead of the covering number of the bayesian nets built on bn.

2.7 Algorithm and theory

The theoretical results presented in the previous section will be used, in the same spirit as covering numbers are used in statistical learning theory, to derive:

- non-parametric non-asymptotic confidence intervals;
- universally consistent algorithms.

Several corollaries are presented in the next sections; after the remark in the previous subsection, these results also hold when hidden variables are involved.

2.7.1 Model selection: selecting BN structures

The first algorithm is concerned with model selection among several bayesian net structures.

Algorithm 10 Model selection

Input: n data points, risk δ, m BN structures $bn_1, \ldots, bn_m$
Initialize $\hat{bn} = bn_1$.
Initialize $bestValue = \inf_{ibn \in bn_1} \hat{L}(ibn) + F(bn_1, \delta)/\sqrt{n}$.
for $i = 2$ **to** m **do**
  $value = \inf_{ibn \in bn_i} \hat{L}(ibn) + F(bn_i, \delta)/\sqrt{n}$
  if $value < bestValue$ **then**
    $\hat{bn} = bn_i$
    $bestValue = value$
  end if
end for
$\hat{ibn} = \operatorname{argmin}_{ibn \in \hat{bn}} \hat{L}(ibn)$
Output: $\hat{ibn}$.

This model selection algorithm first selects the *structure* $\hat{bn}$ by minimizing the empirical error (the minimal empirical error reached for this structure) penalized by a term depending upon the complexity of the structure. The second step is to determine the best bayesian net for this structure, i.e. the one with minimal empirical error.

Corollary C1: *Given the model space* $H = \{bn_1, \ldots, bn_m\}$ *and n data examples, Algorithm1 selects the best Bayes net* $\hat{ibn}$ *built on a model in H up to some approximation error,*

with high probability. Formally, with probability $1-m\delta$ *and for* $\varepsilon = 3\max_i F(bn_i,\delta)/\sqrt{n}$,

$$\forall ibn \in H, L(\hat{ibn}) \leq L(ibn) + \varepsilon$$

Proof:

Let us denote ibn^* the BN with minimal error ($ibn^* = \text{argmin}_{ibn\in H} L(ibn)$) and $\widehat{ibn}_i$ the BN with minimal empirical error among the BN built on bn_i ($\widehat{ibn}_i = \text{argmin}_{ibn\in bn_i} \hat{L}(ibn)$).

The deviation bound on every bn_i states that, for $\varepsilon = 3\sup F(bn_i,\delta)/\sqrt{n}$:

$$P(L(\widehat{ibn}_i) - \hat{L}(\hat{ibn}_i) > \varepsilon/3) \leq \delta$$

Therefore, with probability $1-m\delta$, the above simultaneously holds for $i = 1\ldots m$. As $\widehat{ibn}$ is among the $\widehat{ibn}_i$, it follows that

$$L(\widehat{ibn}) \leq \hat{L}(\widehat{ibn}) + \varepsilon/3$$

By definition of $\widehat{ibn}$ (built on bn_{i_0} and ibn^* (built on bn^*), one has:

$$\hat{L}(\widehat{ibn}) + F(bn_{i_0})/\sqrt{n} \leq \hat{L}(ibn^*) + F(bn^*,\delta)/\sqrt{n}$$

Therefore

$$L(\widehat{ibn}) \leq \hat{L}(ibn^*) + 2\varepsilon/3$$

and finally

$$L(\hat{ibn}) \leq L(ibn^*) + \varepsilon$$

□

This result is the basis for model selection among several BN structures, in the spirit of the celebrated *"structural risk minimization"* [Vap95a].

2.7.2 Parameters learning algorithm: consistency of the minimization of $\hat{L}$

Denoting as before the example distribution as P, considering n examples i.i.d. sampled from this distribution P, noting $\hat{P}$ the empirical distribution and $\hat{L}$ the empirical loss, the standard frequentist approach to learn the bayesian network parameters proceeds as follows. Let *bn* a bayesian network structure, *ibn* an instantiation of *bn*, where $ibn(B)$ denotes the joint probability given by *ibn* for some set of variables B, then the learned $\hat{ibn}$ is such that:

$$\forall i,\ ibn(A_i, Pa(i))/ibn(Pa(i)) = \hat{P}(A_i, Pa(i))/\hat{P}(Pa(i))$$

The approach we propose is based on the minimization of the $\hat{L}$ loss function, i.e,

$$\hat{ibn} = \operatorname*{argmin}_{ibn \in bn} \hat{L}(ibn)$$

Next corollary shows that i) the minimization of the empirical loss function is consistent; ii) the frequentist approach is not consistent.

Corollary C2: *With same notations as above, for any distribution P,*

$$L(\operatorname*{argmin}_{ibn \in bn} \hat{L}) \to \inf_{bn} L$$

For some distributions P, calibrating bn coefficients after $\hat{P}$ asymptotically leads to a non-optimal ibn.

$$L(\{ibn\ s.t.\ ibn(A_i, \mathrm{Pa}(i))/ibn(\mathrm{Pa}(i)) = \hat{P}(A_i, \mathrm{Pa}(i))/\hat{P}(\mathrm{Pa}(i))\} \not\to \inf_{bn} L$$

Proof: The convergence $L(\mathrm{argmin}_{ibn \in bn} \hat{L}) \to \inf_{bn} L$ follows immediately from the fact that the covering number for any ε is finite. The VC-dimension is finite, which implies the almost sure convergence of the empirical loss.

A counter-example is exhibited for the second result, graphically illustrated on Figure 2.7. Let P be the law defined by: $P(A = true \wedge B = true) = a$, $P(A = false \wedge B = false) =$

$1-a$, and $P=0$ otherwise. Assume that $bn=\{P(A),P(B)\}$, i.e., bn assumes the independence of variables A and B.

Calibrating bn after $\hat{P}$ leads to $ibn(A)=\hat{P}(A)\rightarrow a$, $ibn(B)=\hat{P}(B)\rightarrow a$.

Denoting $x=ibn(A),\ y=ibn(B)$, the square loss of ibn is given as:

$$L_2(ibn)=(xy-a)^2+x^2(1-y)^2+y^2(1-x)^2+((1-x)\times(1-y)-(1-a))^2$$

Taking the derivative of the above wrt x or y shows that the derivative is not 0 at the optimum $(x=a,y=a)$ unless $a=\frac{1}{2}$. □

Another limitation of the frequentist approach is related to the estimation of rare events and their consequences, e.g. $P(B|A)$ cannot be estimated if A never occurs. Quite the opposite, the presented approach directly deals with rare events.

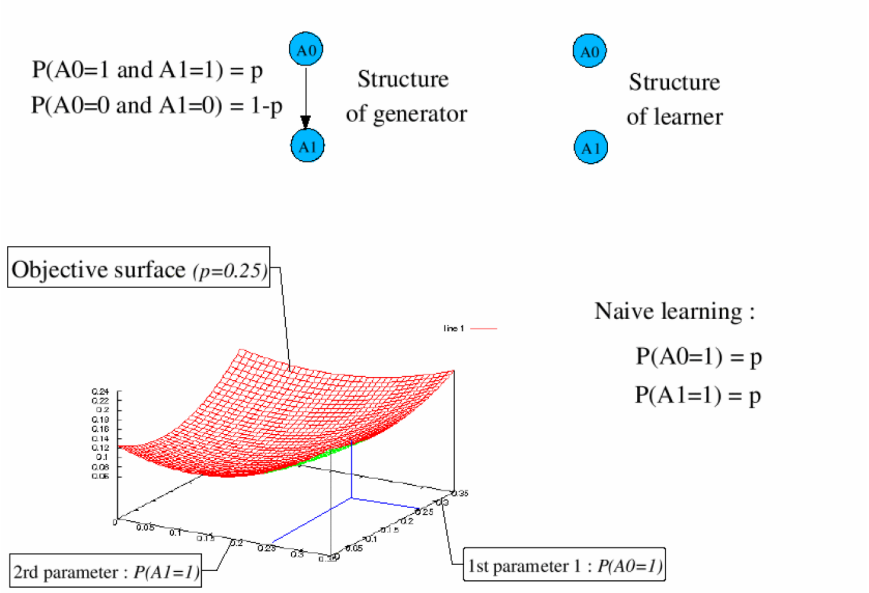

Figure 2.7: Consistency of global optimization. Considering the toy bayesian network (up left: target; up right: learned), the loss function L is represented vs the parameters of the learned BN, respectively $x=P(A_0=1)$ and $y=P(A_1=1)$. The frequentist approach leads to learn $x=y=p$, which is not the global optimum of L.

2.7.3 Universal consistency and bound

Let us assume the existence of some heuristics E, ranking the dependencies among subsets of variables; given the set of variables A_i and their respective parent variables $Pa(i)$, E selects the variable A_i and A_j such that adding A_j to the parent variables $Pa(i)$ results in an admissible bayesian structure.

Such a heuristic, e.g. based on a validation set, can be used for model selection as follows. Noting $E(bn)$ the bayesian structure defined by enriching bn with the additional dependency $A_j \rightarrow A_i$:

Algorithm 11 PAC learning in Bayesian Networks

Input: n data points, risk δ, desired precision $\varepsilon(n) \rightarrow 0$, heuristic E.
Initialize $\hat{bn} = empty\ graph$.
while $F(E(\hat{bn}),\delta)/\sqrt{n} < \varepsilon(n)$ **do**
  $\hat{bn} = E(\hat{bn})$
end while
$\widehat{ibn} = \text{argmin}_{ibn \in \hat{bn}} \hat{L}(ibn)$.
Output: $\widehat{ibn}$, an instantiated bayesian network (structure+parameters).
Bound: $L(\widehat{ibn}) \leq \hat{L}(\widehat{ibn}) + F(\hat{bn},\delta)/\sqrt{n}$.

Corollary C3: *Assuming that heuristic E leads to a bayesian network structure including the probability law in a finite number of steps (*$\inf_{ibn \in bn} L(ibn) = inf_{ibn} L(ibn)$*) ([15]), then*

- *with confidence at least* $1-\delta$, $L(\widehat{ibn}) \leq \hat{L}(\widehat{ibn}) + F(\hat{bn},\delta)/\sqrt{n}$
- $L(i\hat{bn})$ *converges to the minimal loss (*$\inf_{ibn} L(ibn)$*) irrespective of the structure bn, asymptotically with the number n of examples.*

The proof follows from the convergence of $F(bn,\delta)/\sqrt{n}$ to 0 since $F(bn,\delta)/\sqrt{n} < \varepsilon(n) \rightarrow 0$ as $n \rightarrow \infty$.

[15] this is a small and natural hypothesis as the heuristic can simply lead to the complete graph (each variable A_i has $A_j, j < i$ as parents) between observable variables if the number of dependencies is sufficiently large. Note that does not mean that the resulting structure will be optimal, in the sense that it encodes exactly the same dependencies as the probability law.

2.7.4 Universal consistency and convergence to the right network of dependencies

This section is devoted to learning bayesian networks and enforcing two major properties: i) universal consistency (the empirical loss converges toward the minimal loss asymptotically with the number of examples); ii) optimal parsimony, meaning that the size (discussed below) of the network converges toward the optimal one.

The latter property is particularly relevant, as it is difficult to guarantee convergence to a non-redundant structure. Several definitions of "size" can be considered, in relation with various penalization schemes. Under the working hypotheses below, we show that the penalization term can be defined by the user; for specific types of penalization, theorem 8 generalizes the standard results related to *parameter optimality* and *inclusion optimality* [CM02]. Let U denote a function mapping the set of instantiated bayesian network onto the real value space $\mathbb{R}$, such that two bayesian networks based on same structure have same image ($\forall (ibn_1, ibn_2) \in bn\ U(ibn_1) = U(ibn_2)$).

Let $R'(ibn)$ be a function similarly mapping the set of instantiated bayesian network onto the real value space $\mathbb{R}$, user-defined, such that $R'(ibn)$ reflects the complexity of ibn. For example, $R'(ibn)$ can be the number of parameters of the bayesian network (see e.g., 2.7.5).

Let $R(n)$ be a function of n decreasing to 0 as n goes to infinity, and define $R(ibn,n) = R'(ibn)R(n)$.

By abuse of notation, in the following $U^{-1}(n)$ will denote the set of instantiated bayesian networks such that $U(ibn) \leq n$.

Working hypotheses

- H0: for n sufficiently large, $ibn^* \in U^{-1}(n)$;
- H1: $\lim_{n\to\infty} \sup_{ibn\in U^{-1}(n)}\ R'(ibn)R(n) = 0$;
- H2: $\lim_{n\to\infty} F(U^{-1}(n), 1/n^2)/\sqrt{n} = 0$;
- H3: $\lim_{n\to\infty} F(U^{-1}(n), 1/n^2)/(R(n)\sqrt{n}) = 0$;

Under these assumptions, algorithm 11 is extended as follows in algorithm 12.

Algorithm 12 Learning in BN with UC and parsimony guarantees

Input: n data points, application U, application R.
optimize $\hat{L}(ibn)+R(ibn,n)$ under the constraint $U(ibn)\leq n$.
Let $\hat{ibn}$ be one of these optima.
Output: $\hat{ibn}$, an instantiated bayesian network (structure+parameters).

Let us show that the above algorithm is universally consistent and guarantees the convergence toward the optimal structure.

Theorem 8: universal consistency and convergence to the right structure *Let $\widehat{ibn}$ be constructed after algorithm 12, i.e.*

$$\widehat{ibn} \in \underset{U(ibn)\leq n}{\operatorname{argmin}} \hat{L}(ibn)+R(ibn,n)$$

1. **universal consistency:** *if H0, H1 and H2 hold, then $L(\widehat{ibn})$ almost surely goes to the minimal loss L^*;*

2. **convergence of the size of the structure:** *if H0, H1, H2 and H3 hold, then $R'(\widehat{ibn})$ converges to $min\{R'(ibn)\ s.t.\ L(ibn)=L^*\}$.*

Proof:

Define $\varepsilon(bn,n)=sup_{ibn\ s.t.\ U(ibn)=n}|\hat{L}(ibn)-L(ibn)|$. We first prove that algorithm 12 is universally consistent under hypothesis H0, H1, H2.

$$\begin{aligned}
L(\widehat{ibn}) &\leq \hat{L}(\widehat{ibn})+\varepsilon(bn,n)\\
&\leq \inf_{ibn'\in bn} \hat{L}(ibn')+R(ibn',n)-R(ibn,n)+\varepsilon(bn,n)\\
&\leq \inf_{ibn'\in bn} L(ibn')+\varepsilon(bn,n)+R(ibn',n)-R(ibn,n)+\varepsilon(bn,n)\\
&\leq \inf_{ibn'\in bn} L(ibn')+R(ibn',n)+2\varepsilon(bn,n)
\end{aligned}$$

Thanks to H1, we only have to prove that $\varepsilon(bn,n)\rightarrow 0$ almost surely. By definition of $F(.,.)$,

$$P(\varepsilon(bn,n)\geq F(bn,1/n^2)/\sqrt{n})\leq 1/n^2$$

Since for any ε, H2 implies that for n sufficiently large, $F(bn, 1/n^2)/\sqrt{n} < \varepsilon$, it follows that $P(\varepsilon(bn,n) > \varepsilon) \leq 1/n^2$.

Thanks to the Borell-Cantelli lemma, the sum of the $P(\varepsilon(bn,n) > \varepsilon)$ being finite for any $\varepsilon > 0$, $\varepsilon(bn,n)$ almost surely converges to 0, which concludes the proof of universal consistency.

Let us now consider the convergence of the size of the structure.

As stated by H0, $U(ibn^*) = n_0$ is finite. Let us consider $n \geq n_0$ in the following.

$$\begin{aligned}
\hat{L}(ibn) + R(ibn,n) &\leq \hat{L}(ibn^*) + R(ibn^*,n) \\
R'(ibn)R(n) &\leq R'(ibn^*)R(n) + \hat{L}(ibn^*) - \hat{L}(ibn) \\
R'(ibn)R(n) &\leq R'(ibn^*)R(n) + L^* + 2\varepsilon(bn,n) - L(ibn) \\
R'(ibn) &\leq R'(ibn^*) + 2\varepsilon(bn,n)/R(n)
\end{aligned}$$

It remains to show that $\varepsilon(bn,n)/R(n) \to 0$ almost surely, which is also done using Borell-Cantelli lemma. By definition of $F(.,.)$, $P(\varepsilon(bn,n) \geq F(bn,1/n^2)/\sqrt{n}) \leq 1/n^2$.

Hypothesis H3 implies that for any ε and n sufficiently large, $F(bn,1/n^2)/(R(n)\sqrt{n}) < \varepsilon$, and so $P(\varepsilon(bn,n)/R(n) > \varepsilon) \leq 1/n^2$. Thanks to the Borell-Cantelli lemma, the sum of the $P(\varepsilon(bn,n)/R(n) > \varepsilon)$ being finite for any $\varepsilon > 0$, $\varepsilon(bn,n)/R(n)$ almost surely converges to 0. □

We prove optimality for some criterion R'. Let's now see some links to classical optimality criteria.

2.7.5 Links to classical optimality criteria

For a given structure *bn*:

- if there exists a instantiated bayesian network based on *bn* such that *ibn* coincides with the true data distribution P, *bn* is called an I-map of P. Equivalently, all (conditional or marginal) independencies given by *bn* are included in P.

- Inversely, if all (conditional or marginal) independencies of P are in *bn* then *bn* is called a D-map of P.

- if both above conditions are fulfilled, then *bn* is called a P-map of *P*.

One of the objectives when learning the structure of a bayesian network is to have a good approximation of the joint law *P*. However, asymptotically one only needs *bn* to be an I-map of *P*. By definition of ibn^* and after the lemma of section 2.5.2, it follows that ibn^* is an I-map of *P*.

Another and more critical objective is to learn a structure capturing as many independencies of the law as possible. Let $bn \leq bn'$ (respectively $bn < bn'$) denote the fact that the set of laws that can be represented by *bn* is a subset (resp. a strict subset) of the laws that can be represented by bn'; equivalently, every (conditional or marginal) independency of bn' is also part of *bn*.

A structure *bn* which has the minimal number of parameters among the I-maps of *P* is called parameter-optimal. A structure *bn* which is minimal for $\leq$ among the I-maps of *P* is called inclusion-optimal.

When law *P* can be represented by a bayesian network, (that is, there exists a P-map of *P*), it is well known that the parameter-optimal and inclusion-optimal structures coincide and they are Markov-equivalent (section 2.6.4, [Chi02a]).

In this case, by setting $R'(bn)$ to the number of parameters of structure *bn*, H1 holds; therefore *ibn* converges towards ibn^*, a P-map of *P*.

When there exists no P-map for *P*, there can be multiple inclusion-optimal models and multiple parameter-optimal models. But after [CM02] (theorem 2, derived from [Chi02b]), all parameter-optimal models are also inclusion-optimal. Again, setting $R'(bn)$ to the number of parameters of structure *bn*, leads *ibn* to converge towards a parameter-optimal model.

Finally, it is worth noting that the score proposed in section 2.6.3 is *asymptotically consistent* (in the sense of [CM02]): the dominant term becomes the number of parameters as the number of examples goes to infinity.

2.8 Algorithmic issues

The results in the previous sections show that optimizing the empirical loss $\hat{L}$ provides better generalization properties than the usual frequentist approach, at least for some reasonable loss functions.

Unfortunately, in its basic form $\hat{L}$ is difficult to evaluate and to optimize. This section aims at addressing these limitations, through proposing:

- other more practical formulations of $\hat{L}$, and algorithms for computing it (section 2.8.1),
- methods for adapting these algorithms to the computation of the gradient (section 2.8.2).
- optimization methods (section 2.8.3), including adaptive precision (based on estimates of the precision of the computation of the gradient) and BFGS.

Despite these efforts, the proposed criterions are less amenable to optimization than the standard KL-divergence: while the use of L_2-norms avoids some stability problems of the KL-divergence, and is more tailored to the evaluation of expectations, it remains computationally harder and neglects rare events.

2.8.1 Objective functions

The methodology proposed is based on:

- a reformulation of the loss function $\hat{L}$;
- an exact method for the computation of $\hat{L}$;
- a Monte-Carlo method for the approximation of $\hat{L}$;
- a method inspired by the quota method for the computation of $\hat{L}$;

Introduction

Lemma: Let Q denote a probability function over p binary variables, S its L_2 norm ($S = \sum_{i\in\{0,1\}p} Q(i)^2$), n the number of examples noted $x_1, \ldots x_n$, then

$$\hat{L}(Q) = 1 + S + \frac{-2}{n}\sum_{j=1}^{n} Q(x_j)$$

Proof: $\hat{L}(Q) = \frac{1}{n}\sum_{j=1}^{n}\sum_{i\in\{0,1\}p}(Q(i) - \mathbf{1}(x_j)_i)^2$ with $\mathbf{1}(x_j)$ the vector $\mathbf{1}$ representing the example x_j, and $\mathbf{1}(x_j)_i$ the i^{th} coordinate of $\mathbf{1}(x_j)$.

$$\begin{aligned}\hat{L}(Q) &= \frac{1}{n}\sum_{j=1}^{n}\left((Q(x_j)-1)^2 + \sum_{i\in\{0,1\}p, i\neq x_j} Q(i)^2\right)\\ &= \frac{1}{n}\sum_{j=1}^{n}\left(1 - 2Q(x_j) + \sum_{i\in\{0,1\}p} Q(i)^2\right)\\ &= 1 + S + \frac{1}{n}\sum_{j=1}^{n} -2Q(x_j)\end{aligned}$$

□

While term $\sum_{e=1}^{n} -2Q(i_e)$ can be computed linearly wrt the number of examples n and the number of variables p, term S is not as easily dealt with. Other formulations are therefore proposed to compute S in a tractable way.

Remark: The literature has extensively considered the computation of sum of probabilities, e.g. for inference based on marginalization. As S is a sum of squared probabilities, approaches related to bayesian inference and described in [LS88, Coz00, KFL01, GHHS02] can be applied and enforce huge gains in terms of both computational effort and precision.

Properties of the objective function

The objective function $\hat{L}(Q)$, to be minimized, is the sum of the empirical loss $\sum_{j=1}^{n} -2Q(x_j)$ and the quadratic term S. The convexity of $\hat{L}(Q)$ thus depends on the empirical term. While this objective function is not necessarily convex (see a counter-example in Fig. 2.8), it was found to be "almost convex" in all experiments we did.

The computation of the gradient will be examined in section 2.8.2.

Exact method for the evaluation of S

An algorithm based on the decomposition of the product law is proposed in this section to compute term S. The number of required operations depends on the BN structure; the simpler the structure, the lesser the computational complexity of computing S.

This algorithm proceeds iteratively, maintaining two lists of set of nodes respectively called the front F and the back C. During the process, the partial sum of squared probabilities corresponding (related to S) are computed for all assignments of variables in F. F and C are initialized to the empty list. Thereafter, at each time step $t = 1 \ldots p$, F and C

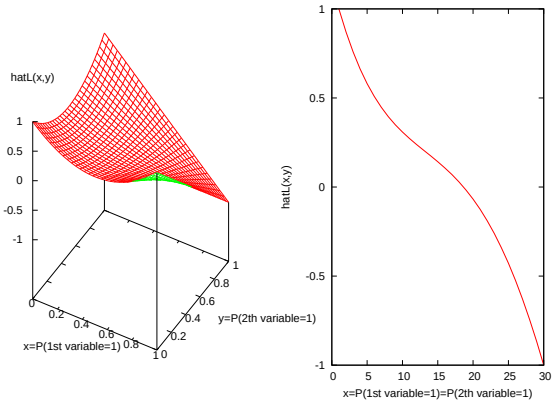

Figure 2.8: Non-convexity of the empirical loss $\hat{L}$. Left: $\hat{L}(x,y)$, where x and y denote the two parameters of a toy BN. Right: $\hat{L}(x,x)$ (diagonal cut of the left hand graph, showing the non-convexity of $\hat{L}$.

are iteratively updated by i) selecting a variable a_t after the rules detailed below; ii) adding a_t to F and updating F accordingly; iii) computing the partial sum of squared conditional probabilities for all assignments of variables in the front.

More precisely, for each time step $t \in 2 \dots p$ let us define:

- F_t is a list of f_t subsets, $F^t = (F_1^t, ..., F_{f_t}^t)$, where F_i^t is a subset of $\{1, ..., p\}$; $F^1 = ()$.
- C_t similarly is a set of f_t subsets C_i^t, for $i = 1 \dots f_t$ where $C_i^t \subset \{1, ..., p\}$; $C^1 = ()$.
- a_t is selected among the last variables (in topological order after the BN structure) such that either they have no successor, or all their successors have been selected in previous steps ($= a_s$ for $s < t$). In case several such a_t exist, a_t is chosen in order to minimize the size of C'^t (see below);
- I_t is the set of indices i such that C_i^t contains a_t, $I_t = \{i / a_t \in C_i^t\} \subset \{1, ..., f_t\}$. I_t can be seen as the set of indices i for which a modification in the computed sums will occur because a variable in the corresponding element of the front depends on the value of a_t (0 or 1). In the following lists, elements of indices $i \notin I_t$ are not be modified between step t and $t+1$, whereas elements of indices $i \in I_t$ include the effect of the variable a_t.

- C'_t is a set of variables, defined as the union of C_i^t for i ranging in I_t, union the parent nodes of a_t ($C'_t = \bigcup_{i \in I_t} C_i^t \bigcup Pa(a_t) \setminus \{a_t\}$).
- C^{t+1} is built from the C_i^t not containing a_t and adding C'_t ($C^{t+1} = (C_i^t)_{i \notin I_t, 1 \leq i \leq f_t} . (C'_t)$ where $a.b$ is the concatenation of lists a and b);
- S^t contains the values of partial sum of squared conditional probabilities for all possible assignments of the variables in the front. An assignment is represented as a function from the set of variables to $\{0,1\}$. $S^t = (S_i^t)_{1 \leq i \leq f_t}$, where S_i^t maintains the partial sums of squared conditional probabilities for every assignment of variables in the i^{th} element of front F^t. Each S_i^t is a function from an assignment to $[0,1]$. S^{t+1} is defined as: $S^{t+1} = (S_i^t)_{i \notin I_t, 1 \leq i \leq f_t} \cdot \left(c' \in 2^{C'_t} \mapsto \sum_{a_t=0}^{1} P(a_t|c')^2 \prod_{i \in I_t} S_i^t(c'_{|dom_i^t}) \right)$ (where $a_{|b}$ is the restriction of a to the domain b)
- F^{t+1} is built from the F_i^t such that $i \notin I_t$ and adding $\bigcup_{i \in I_t} F_i^t \cup \{a_t\}$: $F^{t+1} = (F_i^t)_{i \notin I_t, 1 \leq i \leq f_t} \cdot \left(\bigcup_{i \in I_t} F_i^t \cup \{a_t\} \right)$;
- f_{t+1} is the length of F^{t+1}
- $dom^{t+1} = (dom_i^t)_{i \notin I_t, 1 \leq i \leq f_t} . (C'_t)$

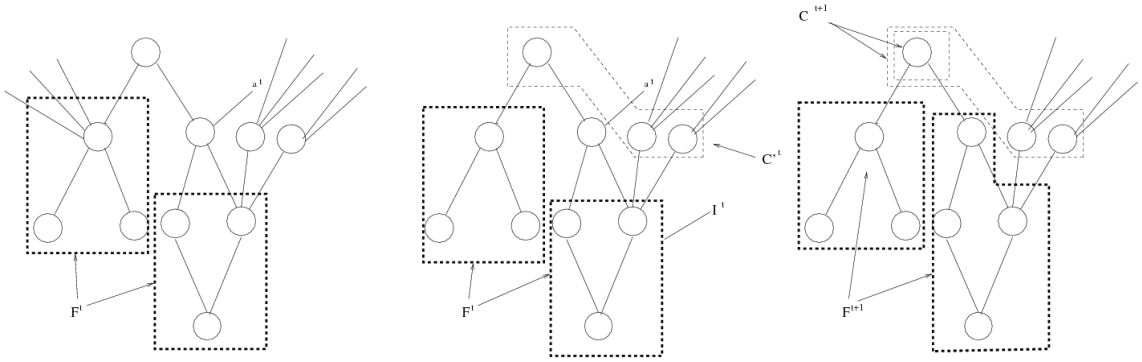

Figure 2.9: Illustration of the exact method algorithm (see text for the definition of each notation).

It is shown by induction that S is equal to the product of the S^t:
For any $t \in \{1, p\}$, for any $1 \leq i \leq f_t$, S_i^t associates to every $c \in 2^{C_i^t}$ (i.e. every assignment

of variables in C_i^t) the sum of squared probabilities of variables in F_i^t conditionally to c. More precisely: $\forall t \in \{1,p\},\ \forall 1 \leq i \leq f_t,\ S_i^t : c \in 2^{C_i^t} \mapsto \sum_{v \in 2^{F_i^t}} P(v|c)^2$ This implies the consistency of the algorithm (see Figure 2.9).

Approximate methods for the computation of S

Because $\hat{L}$ (and likewise its gradient, section 2.8.2) are hard to compute, this section presents an efficient approximation of $\hat{L}$. Since $\hat{L}$ is the sum of the (computationally expensive) term S and a term with fast exact computation (section 2.5.1), the approximation focuses on term S.

This approximation is based on rewriting $S = \sum_{i=1}^{2^p} Q(i)^2$ as $S = EQ(i)$, E being the expectation under the law Q; therefore, this expectation can be approximated using a finite sample drawn according to law Q (Monte-Carlo method) or a refinement thereof (Quota method). Both methods are presented below; they can be used to compute S or its gradient ∇S (see section 2.8.2), both of them being necessary either for the gradient descent or for the BFGS method.

Formally, we shall denote $\widehat{\hat{L}}$ the estimate of $\hat{L}$ where S is replaced by an empirical mean on a finite sample. Likewise, $g = \nabla \hat{L}$ denotes the exact gradient of $\hat{L}$ and $\hat{g} = \widehat{\nabla \hat{L}}$ the approximate gradient.

Monte-Carlo method for the computation of S The Monte-Carlo approach straightforwardly proceeds by simulating the law Q associated to the network and averaging the results. S is thus approximated by $\sum_{j=1}^{n} Q(e_j)^2$ where e_j are i.i.d among $\{0,1\}^p$ with distribution of probability Q.

The variance can be estimated along the same lines. Let us consider the case of $\nabla \hat{L}$ for the sake of readability, the case of $\hat{L}$ being similar. By definition, with d the dimension of the gradient,

$$||\hat{g} - g||^2 = \sum_{i=1}^{d} (\hat{g}_i - g_i)^2$$

Handling $\hat{g}_i - g_i$ as $\frac{\sigma_i N_i}{\sqrt{n}}$ (approximation for the central limit theorem), where N_i is a Gaussian variable with mean 0 and variance 1, and further assuming that the N_i are independent,

it comes:

$$E||\hat{g}-g||^2 = \frac{1}{n}\sum_i \sigma_i^2 EN_i^2 = \frac{1}{n}\sum_i \sigma_i^2 \text{ as } EN_i^2 = 1$$
$$Var||\hat{g}-g||^2 = \frac{1}{n^2}\sum_i \sigma_i^4 Var(N_i^2) = \frac{2}{n^2}\sum \sigma_i^4 \text{ as } Var(N_i^2) = 2$$

It follows that $||g-\hat{g}||^2$ can be upper bounded by the sum of its average and standard deviation, that is $\frac{1}{n}\left(\sum \sigma_i^2 + \sqrt{2\sum \sigma_i^4}\right)$.

Quotas Method for the computation of S A more stable solution is defined as follows. Let us assume that the variables are ordered in topological order, and let us consider the 2^p possible values (assignments) of the whole set of variables, taking in lexicographic order. The probabilities corresponding to those assignments are noted $q_1, q_2, \ldots, q_2p$.

With n the sample size (determining the accuracy of the approximation), let us define $x_i = \frac{i-1}{n}$ for $i = 1,\ldots,n$. For $i = 1,\ldots,n$, let $j(i) \in \{1,\ldots,2^p\}$ be such as $j(i) = \text{argmin}\{j | \sum_{h=1}^{j} q_h \geq x_i\}$. Then S is approximated as $\hat{S} = \sum_{i=1}^{n} q_{j(i)}^2$.

Thanks to the fact that the lexicographic order of assignments is consistent with the ordering of variables in the BN, this sum can be computed very efficiently (for each i, $j(i)$ can be computed in $O(p)$), with the simple procedure described in algorithm 13.

2.8.2 Computation of the gradient

When computing the gradient of $\hat{L}$, the most difficult part is to compute the gradient of S. This section presents an efficient evaluation of ∇S, along the same lines as in section 2.8.1. Consider the following high-level definition of S:

$$S = \sum_j S_j$$

where j ranges over all possible assignments of the p variables. For each j, and the corresponding assignment $(A_1 = a_1,...,A_p = a_p)$, $S_j = P(A_1 = a_1,...,A_p = a_p)^2$. For each j, we define I_j and I'_j the sets of parameter indices appearing in $P(A_1 = a_1,...,A_p = a_p) = \prod_k P(A_k|A_{Pa(k)})$, where each $P(A_k|A_{Pa(k)})$ corresponds to a parameter. If a parameter, with indice i, is involved in $P(A_1 = a_1,...,A_p = a_p) = \prod_k P(A_k|A_{Pa(k)})$, then $i \in I_j$ if $A_k = 1$ and $i \in I'_j$ if $A_k = 0$. We have by definition: $\forall j; I_j \cap I'_j = \emptyset$ and $\forall i; |\{j; i \in I_j \cup I'_j\}| = 1$.

Algorithm 13 Quota Method

Input: number of samples n, a BN.
Initialize $\hat{S} = 0$
for $i = 1$ **to** n **do**
  $x_i = \frac{i-1}{n}$
  Initialize $p = 0$
  Initialize $q = 1$
  for $k = 1$ **to** p **do**
    $q = q \times P(A_k = true | A_{Pa(k)} = a_{Pa(k)})$
    if $x_i - p < q$ **then**
      $a_k = true$
    else
      $a_k = false$
      $p = p + q$
    end if
  end for
  $\hat{S} = \hat{S} + \frac{1}{n} q^2$
end for
Output: $\hat{S}$.

It comes:

$$S_j = \prod_{i \in I_j} p_i^2 \prod_{i \in I'_j} (1 - p_i)^2$$

Then:

$$\frac{\partial S_j}{\partial p_i} = \begin{cases} 0 & \text{if } i \notin I_j \cup I'_j \\ 2S/p_i & \text{if } i \in I_j \\ = -2S/(1-p_i) & \text{if } i \in I'_j \end{cases} \tag{2.7}$$

Therefore, Monte-Carlo method can be adapted as follows to the computation of ∇S:

- draw examples as in the computation of $\hat{L}$;
- for each example, adapt the at most p parameters that are concerned (one per variable).

The computational complexity is that of computing S multiplied by p. The quota

method can be adapted in the same way.

The exact method can be adapted as follows:

- for each parameter p_i of the bayesian network:
  - set the value of the parent variables, such that p_i is relevant;
  - evaluate S for the bayesian network with these fixed values;
  - use equation 2.7 to compute $\partial S/\partial p_i$.

As BFGS can approximate the Hessian very efficiently using successive gradient computations, computing the gradient of S is sufficient to apply the optimization algorithms below.

2.8.3 Optimization

Two optimization algorithms have been considered to minimize $\hat{L}$. The baseline algorithm, based on gradient descent is the simplest approach for non linear optimization. Secondly, we used BFGS, a standard non-linear optimization algorithm which i) is superlinear in many cases; ii) only needs the gradient; as mentioned above it approximates the Hessian thanks to the successive values of the gradient.

We used Opt++ and LBFGSB, freely available on the web, as BFGS optimization softwares. The experimental validation results presented in the next section are obtained with LBFGSB, a limited BFGS algorithm for bound-constrained optimization.

2.9 Experiments

This section is concerned with the experimental validation of the presented approach, starting with the goals of experiments and detailing the empirical results obtained on real and artificial problems, using i) the new learning criteria (section 2.6) and ii) the learning algorithms optimizing these criteria (section 2.7).
All experiments are launched on a PC Pentium IV with 3.0 GHz.

2.9.1 Goals of the Experiments

The experiments are meant to assess both the statistical significance of the presented criteria and the computational complexity of the presented algorithms.

Specifically, a first question regards the quality of the entropy-based criterion (section 2.6), and whether it makes a significant difference compared to the standard criterion based on the dimension of the BN parameters.

A second question regards the efficiency of the presented algorithms. Does BFGS efficiently optimizes $\hat{L}$? More generally, how do the algorithms for the computation of S and ∇S behave? This question will be examined by comparing the exact method with the Monte-Carlo approach, using uniform or quota-based sampling (presented in sections 2.8.1 and 2.8.1), depending upon i) the dimension of the target BN; ii) the size of the training set.

Lastly, our goal is to compare the $\hat{L}$-based learning criterion with the local method (frequentist), and see whether the theoretical superiority of the former criterion is confirmed in practice.

2.9.2 The entropy-based criterion

A theoretical result (theorem 7, section 2.6.3) is that the deviation $L - \hat{L}$ is bounded above by a term depending on the *entropy* of the network besides the standard complexity of the network measured by its number of parameters.

The relevance of this bound is assessed experimentally along the following experimental setting. A bayesian network is randomly constructed as follows:

- a permutation of variables is randomly drawn, uniformly among all permutations;
- for each variable (node) v in this order, and each previous variable w, w is added to the set of parent of v with probability $\frac{1}{2}$;
- each parameter p_i is uniformly drawn in $[0,1]$.

A training set D and a testing set T with respective sizes 2000 and 100000 are generated after the distribution defined by the bayesian network.

For a number $k = 81$ of parameters, $m = 30$ BN structures with k parameters are generated[16]; for each such hypothesis ℓ_i:

- its entropy noted H_i is computed;
- the parameters are learned (by optimizing $\hat{L}$) and the empirical error $\hat{L}_i$ measured on D is recorded;
- the corresponding generalization error L_i is measured on T.

The deviation $L_i - \hat{L}_i$, averaged over the m learners is reported vs k. Fig. 2.10 displays the deviation vs the entropy, for a fixed number of variables (10) and a fixed number of parameters (81). The correlation between the deviation and the entropy is clear, while the entropy term does not appear in the existing learning criteria considered for learning Bayesian Networks. This result successfully validates the merit of the presented criteria.

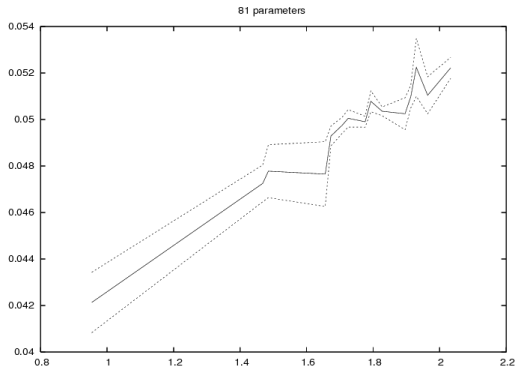

Figure 2.10: Positive correlation of the loss deviation $L - \hat{L}$ averaged on 30 BN ($\pm$ standard error) with the BN entropy, for a fixed number of variables and parameters.

2.9.3 Algorithmic Efficiency

A second objective is to assess the computational efficiency of the algorithms devised for the computation of S and ∇S, namely i) the exact method; ii) the Monte-Carlo method; iii)

[16]The generation process of those structures is the same as described above, with a reject if the number of parameters is not k.

the quota-based method. All three algorithms are first considered in isolation and thereafter inside the optimization loop.

The experimental setting is as follows. 10 bayesian networks are randomly generated with a number of variables ranging in $[20, 50]$, where the i-th node has four parents: two parents are randomly drawn in $[i-6, i-3]$ and the other 2 parents are the $i-2$-th and $i-1$-th nodes.

Preliminary experiments on the approximations of ∇S

Table 2.9.3 displays the results obtained for each algorithm for various problem sizes (number of nodes and sample size).

Algorithm	Time	Relative error
nb nodes=20, sample size = 10000		
Exact	0.68± 0.07	0.
Monte-Carlo	0.04 ± 0.003	0.07 ± 0.02
Quotas	0.04 ± 0.005	0.01 ± 0.003
nb nodes=30, sample size = 10000		
Exact	1.85± 0.15	0.
Monte-Carlo	0.06± 0.004	0.11± 0.02
Quotas	0.06± 0.000	0.05± 0.03
nb nodes=50, sample size = 30000		
Exact	6.26± 0.33	0.
Monte-Carlo	0.29± 0.01	0.19± 0.04
Quotas	0.30± 0.01	0.17± 0.04

These results experimentally validate the exact method as the error is 0 for each run. They also show that the quota-based method improves on the Monte-Carlo method for small problem sizes while it obtains similar results on larger problems. As widely acknowledged for quasi-Monte Carlo methods (see e.g. [Sri00]), their results are often similar to that of vanilla Monte-Carlo approaches in large dimensions.

With respect to the computational effort, both approximate methods are comparable and faster than the exact method by an order of magnitude (factor 17 to 30). Only the

quota-based method will be used in the next section.

Optimization through the approximate computation of S and ∇S

The next objective is to show that the optimization algorithm based on the approximate methods above is reliable. Formally, the algorithmic goal is to determine the optimal parameter values through minimizing $\hat{L}$ (the objective function, section 2.5.1). The minimization is done using BFGS, based on the approximation of S and ∇S using the quota-based approximation algorithm. $\widehat{\nabla \hat{L}}$ and $Var(\widehat{\nabla \hat{L}})$ respectively denote the estimates of $\widehat{\nabla \hat{L}}$ and its variance through the quota method.

The algorithm computes $\hat{\hat{L}}$ (the approximation of $\hat{L}$) using the quota method; and it optimizes it using BFGS. The number of samples (used in the quota method) is increased when the variance $Var(\widehat{\nabla \hat{L}})$ is too high, typically when

$$Var(\widehat{\nabla \hat{L}}) \geq \alpha \times ||\widehat{\nabla \hat{L}}||^2$$

for $\alpha = 0.1$.

The approximation-based algorithm is compared to the exact algorithm, where the exact method is used to compute $\hat{L}$ and $\nabla \hat{L}$.

Note that the structure of the target BN (section 2.9.3) favors the exact method as the network has limited width (each node has 4 parents, whereas there are up to 50 variables).

Figure 2.11 displays the comparative results of the BFGS optimization based on the approximate and exact computation of $\hat{L}$ and $\nabla \hat{L}$. Thanks to the particular structure of the network, the exact method remains very efficient even for 30 nodes, while the approximate method is significantly faster than the exact one. Indeed, in the general case, the exact approach becomes intractable and only the approximate method can be used.

As a proof of concept, these experiments demonstrate the practical efficiency of the proposed approach, based on the BFGS optimization of the approximate loss and its gradient, for small size problems. As noted in section 2.8.1, the adaptation of inference algorithms at the state of the art would be needed to handle large size bayesian networks in the presented framework, and will be considered as a perspective for further research.

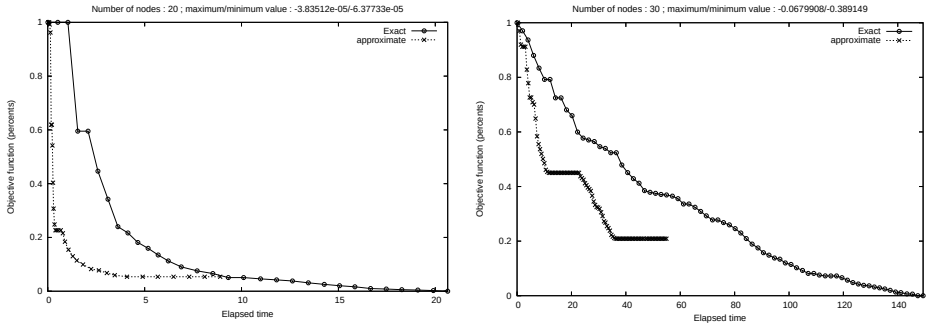

Figure 2.11: BFGS optimization of $\hat{L}$ using exact and approximate computation of $hatL$ and $\nabla\hat{L}$, versus computational time. Left: 20 nodes; Right: 30 nodes, with same target BN as in section 2.9.3.

2.9.4 Loss-based learning

A last objective is to assess the relevance of the proposed criterion and learning algorithms wrt the minimal prediction loss. This objective is empirically investigated on 10 artificial problems and considering 50 randomly generated BN structures $h_1, \ldots h_{50}$. For each problem:

- The target distribution tc is randomly defined as a bayesian network, involving 10 variables and where the DAG is generated as in 2.9.3.
- A training set of n examples is generated after tc, where n ranges in $[100, 900]$.
- The h_i parameters are optimized based on the training set, using the learning algorithm detailed in the previous section.

Taking inspiration in [PA02], the performance criterion is set to the regret, namely the difference between the prediction loss of h_i^* minus the prediction loss of the true target concept.

The results, displayed in Figure 2.12, show that on average parametric learning based on the $\hat{L}$ optimization decreases the prediction error by a factor 2 compared to naive (frequentist) learning. Note that this gain does not rely on structural learning (optimizing the DAG structure too), which is usually one or several orders of magnitude more expensive than parametric learning.

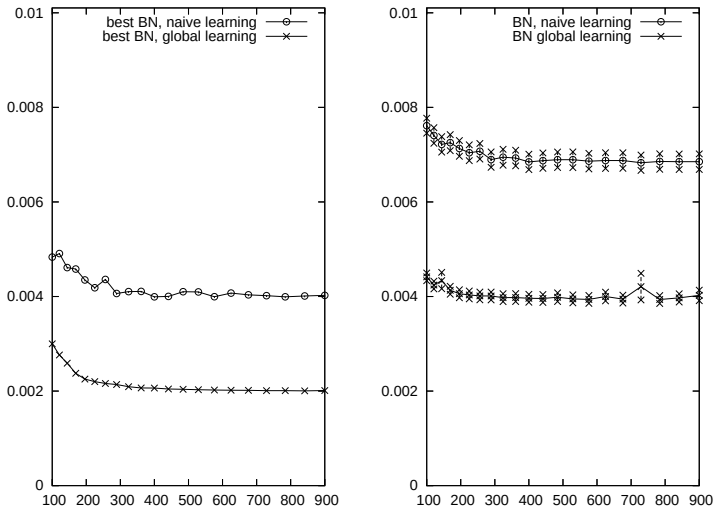

Figure 2.12: Regret (Generalization error - best possible generalization error: $L - L(g)$ with g the BN generating the data.) vs the number of training examples, averaged over 10 problems and 50 hypotheses for each problem. Left: best result for naive (frequentist) and global (our method) learning; Right: average over learners.

2.10 Conclusion

The theoretical and empirical contributions presented in this chapter relate to the structural and parametric learning of bayesian networks.

- As an alternative to the standard frequentist approach, we have proposed a new learning criterion based on the prediction loss L, with some evidences of its relevance for some applications (section 2.2).

- In relation with the prediction loss (more specifically, in order to bound the deviation between the empirical and expected loss), the covering numbers of bayesian networks have been revisited. New bounds involving both the number of parameters of the BN and its complexity, referred to as *structural entropy*, have been proved (theorem 7, section 2.6). While the number of parameters is widely acknowledged a factor of complexity (in the spirit of the BIC/AIC criteria), to our best knowledge the structural entropy term is original. Furthermore, despite the fact that statistical learning theory classically derives loose bounds, the relevance of the structural entropy term has been

empirically evidenced (section 2.9.2).

- The minimization of the empirical loss deviation, defines a parametric learning framework which improves on the standard frequentist approach, both theoretically (section 2.7.2) and experimentally (section 2.9.4). Further, this same framework, increased in the spirit of the structural risk minimization [Vap95a], can be used for structural learning; an algorithm with universal consistency guarantees, enforcing the a.s. convergence towards a minimal size structure, has been presented (theorem 8).

While the optimization of this new learning criterion provides better theoretical guarantees, it is clearly more computationally expensive than the frequentist approach. Therefore, a second contribution of the chapter relates to the algorithmic aspects and tractability of the presented approach. One exact algorithm, taking advantage of the structure properties, and two approximate algorithms (based on Monte-Carlo and quotas methods) have been proposed in order to compute the empirical loss and its gradient (section 2.8). The accuracy of these approximations has been empirically investigated, and a proof of concept showing the good behavior of a BFGS optimization algorithm based on these approximations has been conducted on small size bayesian networks (section 2.9.3).

The presented approach opens several avenues for further research:

- While the bounds based on statistical learning theory are much too conservative in the general case, they often suggest appropriate penalization policies. In our case, regularizing the loss function based on the R term (number of parameters of the network) and the entropy $H(r)$ term appears to be relevant. Formally, the theoretical results in section 2.6 suggest the use of $R(1+H(r)/\ln(1/\varepsilon))$ as regularization, where ε is an estimate of the precision.

- The presented approach deals with datasets where some conjunctions of parents are not present (then $P(A_i|A_{Pa(i)})$ is not defined for some values of the parents $Pa(i)$).

- The presented approach deals with latent variables without requiring any modification, avoiding the use of EM-based approach to estimate the distribution of the latent variables.

- As opposed to standard approaches, the weakness of the proposed approach regards the precise estimate of specific states (e.g. whether some variable assignment occurs with probability 10^{-3} or 10^{-7}), which is better addressed e.g. using KL-divergence. In contrast, the strength of the proposed approach concerns the estimation of macro-events. For instance let us consider the overall fault probability, summarizing the many possible different faults, each of which being modeled by one binary random variable. Characterizing the overall fault probability can be handled with the presented approach.

- Last but not least, the scalability of the presented approach in order to learn large size bayesian network will be considered using adaptations of existing inference algorithms. Indeed, inference algorithms are designed to compute efficiently (taking advantage of the network structure) sum of probabilities (marginalization), while the S term is a sum of squared probabilities. Those issues are known under the factor graphs and sum-product terminologies [AM97, KFL01]. Bayesian inference works described in [LS88, Coz00, KFL01, GHHS02] can be applied and enforce huge gains in terms of both computational effort and precision.

Chapter 3

Robust Dynamic Programming

This chapter focuses on Stochastic Dynamic Programming. As pointed out by Sutton and Barto [SB98], one of the two main threads in reinforcement learning "concerns the problem of optimal control and its solution using value functions and dynamic programming". Dynamic Programming addresses three interdependent objectives: learning the value function, optimizing the action, and sampling the environment.

The central claim of the chapter is that all three objectives must be considered in a integrated way; the overall result depends on the integration of the learning, optimization and sampling modules as much as on their standalone performance.

The contribution of the chapter is twofold. On the one hand, a repository of some prominent algorithms developed for learning, optimization and sampling has been defined and integrated in an open source platform called *OpenDP*[1]. Likewise, a repository of benchmark problems with controllable complexity and inspired from diverse real-world problems, has been developed. Principled experimentations, investigating the combined behaviors of the considered algorithms on these benchmark problems, provide extensive observations about when a method should be used, and how to best combine methods.

On the second hand, a theoretical study of role of randomness in sampling methods is presented.

The chapter is structured as follows. Section 3.1 presents and discusses Stochastic Dynamic Programming. Section 3.2 describes the repository of algorithms and benchmark

[1] OpenDP, `http://opendp.sourceforge.net`

problems integrated in the OpenDP platform. Section 3.3, 3.4 and 3.5 detail results respectively about the optimization, learning and sampling steps.

3.1 Stochastic Dynamic Programming

As mentioned in Chapter 1, at the core of many reinforcement learning and control approaches is the value function V (respectively Q), approximating the expected cost or reward for a given state (resp., a pair (state, action)). The value function, anticipating the potential benefits associated to a state or an action in a given state, is the cornerstone of control methods; note that a direct approach (optimizing a sequence of actions) has exponential complexity in the number of time steps, aka horizon.

Stochastic Dynamic Programming (SDP), briefly introduced in Chapter 1 (the interested reader is referred to [SB98, BT96, Roy01] for a comprehensive presentation), one of the oldest techniques used to compute optimal strategies, is used in many real-world applications with significant economic impact, such as power supply management or energy stock management. It turns out that the SDP approaches have not been thoroughly investigated in RL or in the neighbor community of approximate-dynamic-programming (ADP), despite the fact that SDP techniques might open to many more industrial realizations of reinforcement learning. Its main weakness, a poor scalability wrt the size of the domain, is addressed through learning techniques (see below) or ad hoc domain restriction heuristics [BBS93, SB98].

Let us recall the main equation of Dynamic Programming, already introduced in Chapter 1. Slightly different, though still standard, notations are used to better emphasize the points studied in this Chapter.

Let f denote the dynamic system, s_t the state at time step t, a_t the decision taken at time step t, and c_t the instantaneous cost function. $c_t(s_t, a_t)$ gives the instantaneous cost for taking decision a_t in the state s_t, and $f_t(s_t, a_t)$ is the state reached when taking decision a_t in the state s_t. State $f_t(s_t, a_t)$ and cost $c_t(s_t, a_t)$ can be random variables (at t, s_t and a_t fixed). Only finite horizons will be considered in this Chapter, $t = 1 \dots T$.

Stochastic dynamic programming is based on equation 3.1, describing the backward

through time computation of the value function V:

$$V_t(s_t) = \inf_{a_t} E[c_t(s_t,a_t)] + E[V_{t+1}(f_t(s_t,a_t))] \quad (3.1)$$

Indeed, this equation can not be applied directly to construct the value function; for one, the state space is usually infinite. We will use in the following a particular SDP technique, also known as fitted value iteration, involving learning techniques, building $V_t(s)$ based on a finite sample $(s_i, V_t(s_i)), i = 1 \ldots n$. Basically, SDP thus involves three core tasks:

- Sampling, i.e. selecting the examples $(s_i, V_t(s_i))$ used for learning;
- Regression, i.e. actually constructing $V_t(s)$ based on the available sample;
- Optimization, i.e. solving the above equation wrt a_t, assuming that s_t and an approximation of V_{t+1} are known.

This chapter focuses on the interdependencies of these three tasks, which will be detailed respectively in sections 3.3, 3.4 and 3.5. Let us first discuss the main pros and cons of SDP, and the underlying random process.

3.1.1 Pros and Cons

Compared to other approaches (Chapter 1), the pros of the SDP approach are: i) its robustness with respect to the initial state; ii) a stable convergence by decomposition among time steps (as opposed to fixed point iterations); iii) learning is based on noise-free examples, up to the optimization accuracy and the precision of $V_t(.)$.

The SDP applicability is limited as it relies on the following requirements:

- Cost function c_t must be known. The cost function might be modeled based on the physics of the application domain, or learned from data (see Chapter 2)[2];
- the transition function must be known, be it either modeled based on the physics of the application domain, or learned from data[3];

[2]While no analytical characterization of the cost function is required, one must be able to compute the cost associated to any pair (state, action), independently of any trajectory actually visiting this state.

[3]Again, no analytical characterization of the transition function is required; still, one must be able to simulate the transition function at any state of the state space.

- the value function must be learned offline;
- the approach is not *anytime*, i.e. an early stopping of the process does not give any guarantee about the quality of the current result.

When these assumptions are not satisfied, SDP is not competitive compared to model-free online algorithms, e.g. $TD(\lambda)$ algorithm (Chapter 1). However, it must be emphasized that these requirements are satisfied in quite a few important application domains[4].

Before describing the core sampling, learning and optimization aspects of SDP, let us discuss stochastic aspects thereof.

3.1.2 Random Processes

We use the term *random process* to denote the model of the exogenous part of the transition function, ie the part of the state independent of the action. In a control problem involving a power plant, the exogenous part would be the weather: the weather is part of the state, as for example the demand will depend on the weather. The weather can be predicted from past scenarios, but the weather does not depend on the decisions we can take.

As another example, the "Arm" problem involves a ball to be caught by a controlled arm. The random process is the description of the ball moves; a scenario, i.e. an example or sequence of observations, is a trajectory of the ball. The ball movement does not depend on the action of the arm.

The random process is learned by assuming that trajectories are independent from each other. Let S_i^j denote the state of the random process at time step j in the i-th scenario. Let R_t denote the state of the random process at time step t; the predicted state at time $t+1$, R_{t+1}, is estimated by retrieving the trajectory $S_k^{\cdot}$ closest to R_t :

$$R_{t+1} = S_{t+1}^k \text{ with } k = \operatorname*{argmin}_{\ell} |S_\ell^t - R_t|$$

In the ball example, that means that from the records of the ball trajectories, and for a given time t, to predict the position of the ball at time $t+1$ we look for the scenario in

[4]For instance, in applications such as power supply management, the model of the physical phenomenon is available, including the cost and the transition functions.

which the position of the ball was the closest to the current ball position R_t. With S^k this scenario, the predicted ball position is S^k_{t+1}, i.e. the position the ball had in the chosen scenario.

While the modelling of the random process can (and should) be done in a much more sophisticated way, this aspect will not be considered further in the following, as our focus is on the optimization, Bellman value learning and sampling techniques in SDP.

3.1.3 The core of SDP

Referring to equation 3.1, we shall call:

- *Optimization*, the computation of the $\inf_{a_t}$ for known s_t and $V_{t+1}(.)$. So the "optimization" in the following refers to real function optimizations, but in a SDP framework (optimization at one step will has consequences over other time steps);
- *Learning*, the regression of the Bellman function, i.e. the approximation of the $V_t(.)$ from examples (s^i_t, y^i_t), with $y^i_t = \inf_{a_t} c(s^i_t, a_t) + EV_{t+1}(f(x^i_t, a_t))$. It is supervised regression learning but again in a SDP framework;
- *Sampling*, the choice of the s^i_t used for learning, and from which the optimization is performed.

Optimization

Equation 3.1 is intensively used during a dynamic programming run. For T time steps, if N examples are required for efficiently approximating each V_t, $T \times N$ optimization problems are solved. Note that the optimization objective function is not necessarily differentiable (as complex simulators might be involved in transition function f) and not necessarily convex; moreover, the optimization problem might be a mixed one, involving continuous and boolean variables (e.g. power plant management involve boolean variables).

The main requirement on the optimizer regards its robustness, meant as its worst-case performance. Indeed, due to the many iterations of eq. (3.1), an optimization failure at some time step can hinder the SDP process more severely than many small errors.

Learning

Reinforcement learning commonly involves regression algorithms, as opposed to interpolation or discretization-based learning techniques.

Learning in the RL framework, and more specifically in the SDP framework, brings in new quality criteria and constraints compared to standard supervised learning. In particular:

- Many learning steps have to be performed and the computational cost of one example might be high (at it often involves one simulation step, and can possibly require many simulation steps if it involves the computation of some expectation). Therefore the learning algorithm must be able to learn efficiently from few examples.

- Another critical requirement for the learning algorithm is robustness, in the sense of the L_∞ approximation error; indeed, learning errors can create false local minima, which will be back propagated through time. Accordingly, the standard mean square error criterion, classically minimized by regression algorithms, is not relevant here. If the dynamics of the environment are smooth, this requirement can be unnecessary [Mun07].

Sampling

As pointed out in e.g. [CGJ95b], the ability of the learner to select examples and modify its environment in order to get better examples is one of the key sources of efficiency for learning systems. Such efficient learning systems typically involve (i) a sampler or active learner, that chooses instances in the domain, and has them labelled by some oracle (e.g. the domain expert or the environment); and (ii) a passive learner that exploits the examples, i.e the pairs (instance, label), to construct an estimation of the oracle (target concept in the classification case, target function in the regression case).
Various forms of active learning have been proposed:

1. Blind approaches include methods which do not take into account the labels [CM04], and only select instances that are well distributed in the domain, e.g. in a quasi-random manner; here, the learning process is a three-step process: i) sample the instances; ii) call the oracle to label them; iii) learn the target concept/function;

2. Non-blind approaches, in which the sampler uses the former examples in order to choose the next instances to be labelled; possibly, the passive learner is embedded in the active learner or sampler. Non-blind approaches use various criteria; [LG94] selects instances where the current estimation of the target function reaches a maximum degree of uncertainty; [SOS92] chooses instances that maximally decrease the size of the version space. Other approaches include [SC00], with application to Support Vector Machines (SVM), and [CGJ95a], with application to Neural Nets.

3.2 OpenDP and Benchmarks

The section presents the Reinforcement Learning Toolbox[5] called **OpenDP**, together with the benchmark problems considered in the extensive experimental study detailed in next section.

3.2.1 OpenDP Modules

For the sake of generality, *OpenDP* handles continuous, discrete or mixed state and action spaces. Although any kind of controller can in principle be integrated in *OpenDP*, the main focus has been on Stochastic Dynamic Programming (SDP), motivated by the real-world industrial applications using such an algorithm and their economic and ecological impact.

The *OpenDP* Toolbox integrates many existing libraries together with our own implementations, Its contents is decomposed into optimization, learning and sampling components. Additionally, a visualization module (Appendix A) enables to visually inspect a value function, compare the algorithm performances, assess the confidence of the optimization using the Veall test [MR90], and so forth.

[5] *OpenDP* is an opensource software under GPL license, freely available at `http://opendp.sourceforge.net/`. OpenDP is written in C++, using Qt library. It works under Linux and Windows operating systems (Fig. A.4).

Optimization module

The optimization module includes gradient based methods, evolutionary computation algorithms, and Monte-Carlo or quasi Monte-Carlo methods. All methods are available as plugins of existing libraries, or have been developed specifically for *OpenDP*:

- Gradient based methods include:
  * an implementation of the BFGS algorithm in the LBFGS library [BNS94, BLN95];
  * several implementations of quasi Netwon methods in the Opt++ library (`http://csmr.ca.sandia.gov/opt++/`);

- Evolutionary algorithms include:
  * Open Beagle, developed by Christian Gagné [Gag05] (`http://beagle.gel.ulaval.ca/`);
  * Evolutionary Objects, developed in the EvoNet framework [KMRS02] (`http://eodev.sourceforge.net/`);
  * a home-made Evolutionary Algorithm;

- Quasi-Random methods involve low discrepancy/low dispersion sequences [LL02] (more on this in section 3.5);

- Lastly, the Derivative-free optimization algorithm CoinDFO, which builds a model of the objective function (`http://projects.coin-or.org/Dfo`).

Learning module

The learning module includes the main regression libraries and algorithms available at the time of writing:

- Weka [WF05] is the most comprehensive machine learning library, and one of the most used (`http://www.cs.waikato.ac.nz/ml/weka/`). It includes dozens of regression algorithms of all kinds;

- Torch (`http://www.torch.ch/`) includes Support Vector Machine algorithms, Neural Networks, and K-Nearest Neighbors;

- Fast Artificial Neural Network Library (FANN) (`http://leenissen.dk/fann/`) is a comprehensive suite of learning algorithms for neural networks.

Sampling module

OpenDP includes several sampling methods, which will be detailed in section 3.5:

- based on low discrepancy sequences (section 3.5.4);
- based on low dispersion sequences (section 3.5.4);
- active sampling methods (section 3.5 and following).

3.2.2 Benchmarks problems

A suite of benchmark problems is integrated in *OpenDP*, making it easy to systematically assess the algorithms presently included in the Toolbox and the future ones. The suite is meant to comprehensively illustrate different algorithmic issues (Table 3.2.2). It involves diversified problems inspired from real-world ones; specifically, the two stock management problems are inspired from real-world power plant applications. Lastly, the dimension of almost every problem is tunable in order to effectively assess the scalability of every applicable method and heuristics[6].

All problems involve continuous state and action spaces. Some problems involve an exogenous random process (section 1); the state space (involved in the learning of the value and transition functions) is thus decomposed into two parts, the state space *stricto sensu* and the model of the random process (which is not controllable by definition). It is worth noting that none of these problems is trivial, i.e. they all defeat greedy strategies (selecting the action that optimizes the instantaneous cost in every time step does not work).

The seven problems in the suite are detailed below with their difficulties.

- **Stock management**: aims at the optimal use of k stocks to satisfy the demand ($k = 4$ in the baseline experiments). The state space is $\mathbb{R}^k$ (the stock levels). The action

[6]The interested reader will find the code and parameters in the *OpenDP* source code, and a basic manual to run the experiments.

Name	State space dimension (basic case)	Random process dimension	Action space dimension (basic case)	Number of time steps	Number scenarios
Stock Management	4	1	4	30	9
Stock Management V2	4	1	4	30	9
Fast obstacle avoidance	2	0	1	20	0
Many-obstacles avoidance	2	0	1	20	0
Many-bots	8	0	4	20	0
Arm	3	2	3	30	50
Away	2	2	2	40	2

Table 3.1: The *OpenDP* benchmark suite, including seven problems. Columns 1 to 3 indicate the size of the state, random process state and action spaces. Column 4 indicates the time horizon and Column 5 is the number of scenarii used to learn the random process.

space is $\mathbb{R}^k$, describing for each stock the positive/negative (buy/sell) decision. The random process is the (exogenous) demand. The cost function is a smooth one, aggregating the price of the goods and penalties when the demand is not satisfied. The difficulty of this problem lies in the learning and optimization steps: accurately estimating the cost function and precisely optimizing the value function.

- **Stock management V2**: is very similar to the previous problem, but easier; it involves different constraints and cost function.

- **Fast obstacle avoidance**: aims at controlling a bot in order to reach a target position while avoiding a (single) obstacle. The bot speed is constant; the action is the direction to be taken by the bot (continuous, in $[0, 2\pi]$). The state space $\mathbb{R}^2$ describes the bot position; there is no detection of the obstacle (ie the position of the obstacle has to be learned and modelled through the Bellman value function). The cost function aggregates the time needed to reach the target and (large) penalties if the bot hits the obstacle. This problem is is a rather simple one; the difficulty lies in the identification of the obstacle position, relying on the learning and sampling steps.

- **Many obstacles avoidance**: is similar to the previous problem, but more difficult as it involves many obstables.

- **Many-bots**: aims at controlling k bots in a 2D map involving simple, not moving, obstacles ($k = 16$ in the baseline experiments). The state space is $\mathbb{R}^{2k}$ (the 2D coordinates of each bot). The action space is $[0, 2\pi]^k$ (the direction angle of each bot; the bots have same constant speed). The cost function aggregates i) the sum of the distances between each bot (aiming at a flock behavior where the bots are close to each other); ii) their distance to a target position; iii) penalties for obstacle hitting. The main difficulty comes from the cost function, including many local optima (when bots are close from each other, but far from the target) in a large state space. This heavily multi-modal landscape thus makes the optimization step challenging.

- **Arm**: aims at controlling an arm to reach a moving target. The arm is made of k connected segments in a 2D plane ($k = 2$ in the baseline problem). The first segment has one extremity fixed at position $(0,0)$, and can make a free 2π rotation. Other segments have their rotation in $[-\pi/2, +\pi/2]$. The state space thus is $[0, 2\pi] \times [-\pi/2, +\pi/2]^{k-1}$. The target is a random process in $\mathbb{R}^2$. The cost is computed from the distance d between the moving extremity of the arm and the center of the target, and defined as $min(d, C)$ where C is a constant. The main difficulty of the problem lies in the cost function; its structure and the value of constant C create large optimization plateaus (where the cost is constant) where greedy optimization can but wander hopelessly. The challenge is on the learning step, that must be very efficient and allow an efficient propagation back through time. An additional difficulty comes from the interdepencies and constraints among the segment angles, specifically challenging the optimization step.

- **Away**: is similar to the previous problem with two differences: i) the goal is to avoid the target instead of reaching it; ii) the target is moving much faster.

3.3 Non-linear optimization in Stochastic Dynamic Programming (SDP)

This section focuses on the thorough evaluation of optimization methods in the Stochastic Dynamic Programming framework. We first briefly review the state of the art, before discussing the main aspects of "robust optimization" in relation with SDP. The optimization algorithms involved in *OpenDP* are described and comparatively assessed on the benchmark suite.

The main contribution of this section lies in the extensive empirical assessment presented. While many papers have been devoted to Approximate Dynamic Programming [PP03, Cai07, Pow07, EBN06], some papers only report on applying one algorithm to some problems; when comparing several algorithms, some papers consider problems favoring one of the algorithms. Furthermore, most papers discard the optimization aspects, sometimes handled through discretization and sometimes just not discussed.

The goal of this section thus is to show the major impact of the optimization step on the overall SDP result, and to understand where the real difficulties are.

3.3.1 Introduction

Large scale SDP applications, such as energy stock-management, define continuous optimization problems that are usually handled by traditional linear approaches: i) convex value-functions are approximated by linear cuts (leading to piecewise linear approximations (PWLA)); ii) optimal decisions are computed by solving linear problems. However, this approach does not scale up as the curse of dimensionality badly hurts PWLA. New approaches, specifically new learning methods must thus be considered. As a side effect, as the Bellman function is no longer a convex PWLA, the action selection task concerned with minimizing the (expectation of the) cost-to-go function, can no longer be achieved using linear programming.

The continuous action selection task thus defines a nonlinear programming problem. The problem of non-linear optimization for continuous action selection, which has received

little attention to our best knowledge[7], is the focus of this section. We compare several non-linear optimization algorithms together with discretization-based approaches, to assess the impact of the action-selection step in the SDP framework.

3.3.2 Robustness in non-linear optimization

"Optimization objective function" and "fitness" are used interchangeably in this section.

While robustness is perhaps the most essential demand on non-linear optimization, it actually covers very different requirements:

1. Stability. The result x should be such that i) x fitness is satisfactory; ii) the fitness is also satisfactory in the neighborhood of x. Stability is a very important requirement e.g. in optimal design. Optimization stability has been extensively discussed in the field of *Evolutionary algorithms*; in particular, [DeJ92] argues that *Evolutionary algorithms* are interested in areas with good fitness, more than in function optimization *per se*.

2. Avoidance of local minima. Iterative deterministic methods such as hill climbing are prone to fall down in local minima; multiple restart techniques (relaunching the algorithm with a different starting point) are commonly used to overcome this drawback.

3. Resistance to fitness noise. Various models of noise and an extensive discussion about noise-related robustness can be found in [JB05, SBO04, Tsu99, FG88, BOS04].

4. Resistance to unsmooth landscape. Gradient-based methods hardly handle unsmooth or discontinuous fitness functions, e.g. due to the use of penalties. In contrast, *Evolutionary algorithms* most often depend on fitness ranks (i.e. the construction of new instances only depends on the ranks of the current instances in the population, as opposed to their actual fitness values), and are therefore invariant under monotonous fitness transformations [GRT06]. In practice, *Evolutionary algorithms* make no difference between $||x||^2$ or $||x||$ optimization, whereas the latter fitness severely hinders standard Newton-based methods like BFGS [Bro70, Fle70, Gol70, Sha70].

5. Stability wrt random effects. Many optimization algorithms, even the so-called deterministic ones, involve random aspects (e.g. in the initialization step; when breaking ties). The dependency of the result wrt such random effects, and the probability of a catastrophic

[7] On the one hand few works in the literature deal with continuous action selection [vHW07]; on the other hand these most often focus on linear optimization.

failure, is a major aspect of algorithmic robustness. Population-based algorithm are clearly more robust in this respect than single-point methods (e.g. simulated annealing); still, the design of the initialization routine is acknowledged to be a major factor of efficiency in *Evolutionary algorithms* [KS97].

Interestingly, all above acceptions of the polysemic "Robustness" criterion are relevant in the SDP framework, to diverse extents.

Stability is fully relevant to cope with the imperfections in the learning step. In the Fast-obstacle-avoidance problem for instance, the robot has to avoid obstacles and the strict fitness optimization might lead to trajectories that are very close to the obstacle; when taking into account the inaccuracies of the learning step (in estimating the robot position), "very close" might just be "too close".

Avoidance of local minima is relevant as SDP indifferently considers convex and non-convex optimization problems (e.g. robotics most usually involves non-convex problems). Further, even when the true value function is convex (the law of increasing marginal costs implies the convexity of many stock management problems), its approximation is not necessarily convex.

Resistance to fitness and gradient noise is also relevant. On the one hand, the underlying fitness function is noisy as it is learned from (a few) examples. On the other hand, there is no guarantees whatsoever on the fitness gradient, even if it can be computed (the fact that $||\hat{f} - f||$ is small or bounded does not imply any bound on $||\nabla\hat{f} - \nabla f||$).

Resistance to unsmooth fitness landscape is mandatory: on the one hand, strong discontinuities exist in continuous problems (e.g. as penalties are used to handle obstacle hitting); on the other hand, SDP often meets mixed continuous/integer optimization problems (stock management problems involve integer numbers of production units).

Finally, we consider Resistance to random effects and catastrophic events to be the most essential criterion within the SDP framework. Indeed, SDP handles many thousand (similar) optimization problems in a single run; therefore, if catastrophic events can occur, they will. More precisely, an SDP-compliant optimization algorithm should be able to address all problems in an optimization problem family, and provide results up to 95% precision − as opposed to, address 95% problems in the family with an arbitrarily good precision.

3.3.3 Algorithms used in the comparison

Let us make it clear that none of the algorithms below was devised to address a specific benchmark problem of the *OpenDP* suite. Rather, our goal is to propose a neutral and comprehensive assessment, considering diversified problems (section 3.2.2) and diverse algorithms, including standard tools from mathematical programming, *Evolutionary algorithms* and discretization-based approaches.

Evolutionary algorithms address binary and continuous optimization problems [BHS91, BRS93, Bey01] and can also handle mixed continuous-integer optimization problems (e.g. [BS95]). However, as not many algorithms can handle mixed optimization problems, and for the sake of a meaningful comparison with other methods, the experiments only consider continuous optimization problems.

Besides *Evolutionary algorithms*, derivative-free optimization methods [CST97] and limited-BFGS with finite differences [ZBPN94, BLNZ95] are considered. Naive approaches, random and quasi-random methods are also considered as they are enable frugal any-time optimization. They include discretization techniques, evaluating a predefined set of actions, and selecting the best one. As detailed below, dispersion-based and discrepancy-based samplings will be used to construct the predefined set of actions.

Let us detail the optimization algorithms involved in the comparative experiments. All but the baseline algorithms are existing codes which have been integrated to the *OpenDP* toolbox.

- Random search: randomly draw N instances in the search space, compute their fitness and return the best one.

- Quasi-random search-1: idem, where low discrepancy sequences [Nie92, Owe03] replace uniform sampling. Low discrepancy sequences have been primarily investigated for numerical integration and later applied to learning [CM04], optimization [Nie92, AJT05], and path planning [Tuf96], with significant improvements compared to Monte-Carlo methods. A hot research topic concerns low discrepancy sequences in highly dimensional spaces [SW98, WW97], (with particular successes when the "true" dimensionality of the underlying distribution or domain is smaller than the apparent one [Hic98]) and the use of scrambling-techniques [LL02].

- Quasi-random search-2: idem, using low-dispersion sequences [Nie92, LL03, LBL04]. While low-dispersion is related to low-discrepancy, the former criterion is easier to optimize; noting d the Euclidean distance, the dispersion of a set of points P reads:

$$Dispersion(P) = \sup_{x \in D} \inf_{p \in P} d(x,p) \tag{3.2}$$

  A relaxed criterion (to be maximized) will be considered in the section, defined as:

$$Dispersion_2(P) = \inf_{(x_1,x_2) \in D^2} d(x_1,x_2) \tag{3.3}$$

  Criterion (3.3) is optimized greedily and iteratively; instance x_i is optimized in order to maximize the dispersion of $(x_1,\ldots x_i)$ conditionally to $(x_1,\ldots,x_{i-1})$. This procedure is fast and any-time (it does not require the number of instances to be known beforehand), although it only provides a local optimum wrt equations (3.3) or (3.2).

- Criterion (3.3) is biased toward instances that are the frontier of the search space, contrasting with (3.2); low-dispersion sequences biased against the frontier of the search space are constructed, replacing eq. (3.3) with:

$$Dispersion_3(P) = \inf_{(x_1,x_2) \in D^2} d(x_1, \{x_2\} \cup \partial D) \text{ with } \partial D \text{ the frontier of D} \tag{3.4}$$

  Likewise, criterion (3.4) is optimized in a greedy and iterative way.

- Covariance Matrix Approximation-based Evolution Strategies (CMA-ES) [HO96, KMRS01, Gag05] are *Evolutionary algorithms* dedicated to continuous optimization; the implementations considered are those of the EO [8] and OpenBeagle libraries [9];

- The Hooke & Jeeves (HJ) algorithm [HJ61, Jr.63, Wri] is a geometric local optimization method; the implementation considered is M.G. Johnson's [10];

[8] http://eodev.sourceforge.net/
[9] http://freshmeat.net/projects/openbeagle/
[10] http://www.ici.ro/camo/unconstr/hooke.htm

- Limited-BFGS with finite differences uses an approximated Hessian in order to approximate Newton-steps without the huge computational and memory cost associated to the use of a full Hessian; the implementation used is that of the LBFGSB library [ZBPN94, BLNZ95].

- Finally, we also used a home-made *Evolutionary algorithms*[11] with standard operators, detailed in [TG07] and in Appendix B.1. Two versions have been considered, with and without memory (noted EA and EANoMem), where the former returns the best instance ever constructed, and the latter one, the best instance in the last population.

The initialization of point-based algorithm takes the domain center point as initial instance; population-based algorithms consider a uniformly selected population. All restarts (e.g. when the progress falls below the machine precision) consider a new starting point uniformly drawn.

When relevant, the step size is set to half the domain width (on each axis).

The parameters of every algorithm have been tuned on a same time budget, considering an independent benchmark problem ([SHL+05]), in order to avoid overfitting a particular problem, and to study the "out of the shelf" performance of the algorithms, taking into account the fact that some parameters are more difficult to tune than other ones.

Finally, the DFO-algorithm [CST97] [12] has been discarded due to its huge computational cost; the CMA-ES from Beagle [HO96, Gag05] has been discarded as it is similar to CMA-ES from EO [KMRS01].

Overall, four types of algorithms have been considered:

- 2 gradient-based algorithms (LBFGS and LBFGS with restart);

- 3 evolutionary algorithms (EO-CMA, EA, EANoMem);

- 4 sampling-based algorithms (Random, Quasi-random, Low-Dispersion, Low-Dispersion "far from frontiers" (LD-fff));

- 2 pattern-search algorithms (Hooke&Jeeves, Hooke&Jeeves with restart).

[11] `http://opendp.sourceforge.net`
[12] `http://www.coin-or.org/`

3.3.4 Experiments

After describing the experimental setting, we summarize the general lessons learned from the experiments and finally discuss the strengths and weaknesses of the various algorithms in relation with the specificities of the benchmark problems.

Experimental settings

All problems of the benchmark suite (section 3.2.2) have been considered: in their baseline dimension d (Table 3.2.2); in dimension $2 \times d$; in dimension $3 \times d$; and in dimension $4 \times d$ (increasing both the state and action space dimension).

All optimization algorithms are combined with the same learning algorithm, set to SVMTorch with Laplacian kernel and tuned hyper-parameters [CB01], learning from 300 examples sampled after quasi-random methods in each time step.

Each optimization algorithm is allowed i) a fixed number of points for sampling-based algorithms; ii) 100 function or gradient evaluations for other algorithms. Every result is the average performance out of 66 independent runs.

Summary of the results

As could have been expected, there is no killer algorithm dominating all the other ones. The best algorithm depends on the benchmark problem. A first outcome of this extensive study thus is to show that definite conclusions offered in the literature about the superiority of such or such algorithm "in general", are not reliable. For each particular benchmark, the best algorithm (with 5% significance over all other algorithms, in bold, or with 5% significant over all algorithms except the second best, in italic), is given in Table 3.2. These results must be taken with care; despite the diversity of the benchmark problems, we are still far from being able to tell which algorithm is the best one for a new SDP problem.

This being said, some relatively stable observations are made:

- For sampling-based approaches:
  - Quasi-random methods dominates random search (they are significantly better in 17 out of 20 experiments, and obtain similar performances on the remaining

3 problems);

  - Low-dispersion (LD) methods, biased toward the domain frontier, are the best ones in “bang-bang” problems, such as the ”away” or ”arm” problems where optimal actions are often close to the boundary of the action space. LD dominate random search on 10 out of 20 problems only; in other words, their performance is problem dependent;
  - LD-fff methods, with a lesser bias toward the domain frontier, outperform random search on 14 out of 20 problems (but are far less impressive than LD ones on bang-bang-problems).

- For order-2 methods, including gradient-based mathods and CMA-ES[13]:
  - LBFGSB (respectively Restart-LBFGSB) outperforms quasi-random methods on 9 (resp. 10) out of 20 problems only, which is blamed on three factors: i) the limited number of examples; ii) the non-convex nature of the landscape; iii) the cost of estimating a gradient by finite-differences;
  - CMA-ES is sometimes very efficient; in particular it is the best algorithm for stock-management problems in medium or large dimensions (≥ 8), where the optimization accuracy matters. The main weakness is its computational cost per iteration, generally much higher than for other methods, explained by the estimation of the covariance matrix (similar in spirit to the Hessian); on the other hand, it does not involve any gradient computation and therefore does not suffer from afferent limitations (e.g. unsmoothness);

- Pattern-search methods (the Hooke&Jeeves algorithm with Restart) outperform quasi-random methods on 10 problems out of 20;

- For *Evolutionary algorithms*:
  - CMA-EO (very) significantly outperforms quasi-random methods on 5 out of 20 problems, namely on stock-management problems in high-dimension;

[13] As CMA uses a covariance matrix which is strongly related to the Hessian.

– EA outperforms quasi-random (respectively, random) methods on 14 (resp. 17) out of 20 problems, significantly in most cases. EANoMem outperforms quasi-random (respectively, random) methods on 14 (resp. 15) out of 20 problems. *Evolutionary algorithms* appear to be the algorithms with most stable performances in the range of experiments.

Detailed results for each optimization algorithm and each problem will be found on Appendix B.2.

3.3.5 Discussion

The experimental study presented in this section is meant to be as neutral and objective as possible. The tuning of the hyper-parameters of each algorithm was allowed the same time budget, although some algorithms might require more tuning efforts to reach better performances[14]. Based on these experiments in the domain of Approximate Dynamic Programming, three claims are made:

- **High-dimensional stock-management**. On medium and high-dimensional problems, for landscapes "moderately unsmooth", our recommendation is to use CMA-ES. Although it is less robust than *Evolutionary algorithms* in the general case, its performances are excellent in problems such as stock resource management. CMA-ES, a most celebrated variant of EA, was found to be a very good candidate for non-linear optimization in high-dimensional search spaces provided that the landscape is smooth enough to enable the covariance matrix adaptation. In the range of experiments (d varies in $[4, 16]$), its performances are significantly better than others; in particular, LBFGS is not satisfactory, which was blamed on the fact that convexity or differentiability cannot be reliably assumed in ADP (section 3.3.2).
  The main weakness of CMA-ES is a huge computational cost. The comparison framework allows each algorithm the same number of calls to the fitness function, which makes sense if the computational effort mainly comes from the fitness-evaluations. Otherwise, CMA-ES might be prohibitively expensive.

[14]The reader will find the open source *OpenDP* Toolbox, including all algorithms and benchmark problems, and will be able to experiment with other parametrization/conditions.

Problem	Dim.	Best algo.	QR beats random	EA beats random ; QR	LBFGSBrestart beats random;QR	LD beats random;QR
Stock	4	**LDfff**	y	y;n	y ; n	y ; n
and	8	**EoCMA**	**y**	n;n	n ; n	n ; n
Demand	12	*EoCMA*	**y**	n;n	n ; n	n ; n
	16	**EoCMA**	n	n;n	n ; n	n ; n
Stock and	4	**LD**	**y**	**y**;y	**y**; **y**	**y**; **y**
Demand2	8	EoCMA	n	y;**y**	n ; **y**	y ; **y**
Fast Obst. Avoid.	1	**HJ**	y	y;n	n ; n	**y**; y
Many Obst. Avoid.**	1	EA	y	y;y	y ; y	y ; y
Many bots**	4	**EA**	n	**y**;**y**	n ;n	n ; n
	8	**EANoMem**	y	**y**;**y**	n ;n	n ; n
	12	**LDfff**	**y**	**y**;**y**	n ;n	n ; n
	16	*EANoMem*	**y**	**y**;**y**	n ;n	y ; n
Arm*	3	LD	y	**y**;y	y ; y	**y**; **y**
	6	HJ	**y**	**y**;y	y ; y	**y**; **y**
	9	LD	**y**	**y**;n	y ; y	**y**; **y**
	12	**LD**	**y**	**y**;**y**	y ; y	**y**; **y**
Away*	2	LD	y	y;y	y ; n	**y**; y
	4	LD	y	**y**;y	y ; y	**y**; **y**
	6	LD	y	y;y	y ; y	**y**; **y**
	8	**LD**	y	y;y	y ; y	**y**; **y**
Total			17/20	17/20 ; 14/20	11/20 ; 10/20	14/20 ; 12/20

Table 3.2: Summary table of experimental results. *-problems corresponds to problems with nearly bang-bang solutions (best action near the frontier of the action space). **-problems are those with high (unsmooth) penalties. For the "best algorithm" column, **bold** indicates 5% significance for the comparison with all other algorithms and *italic* indicates 5% significance for the comparison with all but one other algorithms. **y** holds for 10%-significance. LD is significantly better than random and QR in all but one case and appears as a natural efficient tool for generating nearly bang-bang solutions. In **-problems, EA and EANoMem are often the two best tools, with strong statistical significance. Stock management problems (the two first problems) are very efficiently solved by CMA-ES, which is a good compromise between robustness and high-dimensional-efficiency, as soon as dimensionality increases.

- **Robustness requirement in highly unsmooth problems.**. *Evolutionary algorithms* are the only algorithms consistently outperforming quasi-random methods on unsmooth fitness landscapes, e.g. involving penalties (problems with legend ** in Table 3.2). While *Evolutionary algorithms* are not always the best, they almost always

outperform random methods; the celebrated robustness of *Evolutionary algorithms* makes them quite appropriate to ADP, specifically with regards to the Stability in front of random effects (section 3.3.2).

- **Bang-bang problems**. A particular case is that of problems where the optimal action is most often located on the frontier of the action space, referred to as bang-bang control problems. Unsurprisingly, LD methods are the best ones for such problems, dues to their bias toward the domain frontier. A relevant any-time strategy for finding efficient bang-bang controllers thus appears the LD method; depending on the allowed number of function-evaluations, LD samples the center, the corners, the border and thereafter the whole action space. Note also that LD is among the worst strategies for non bang-bang problems; some prior knowledge is thus required to decide whether LD is a good candidate.

3.4 Learning for Stochastic Dynamic Programming

This section focuses on the thorough evaluation of learning methods, used to estimate the Bellman value functions in the Stochastic Dynamic Programming framework. We first briefly discuss the state of the art, and list the learning algorithms integrated in the *OpenDP* framework. These algorithms are comparatively assessed on the benchmark suite (section 3.2.2).

As in section 3.3, the main contribution of this section lies in the extensive empirical assessment of learning algorithms in the SDP framework. As pointed out in [KR95], learning-based dynamic optimization is not yet widely used for industrial applications; one reason why learning-based DP remains confined in research labs seems to be the lack of clear expertise about which algorithms should be used and when. Many authors implement one algorithm and compare its results to the state of the art on a single problem, often considering different parametrization, different loss functions, different time schedules, and different action spaces (as many algorithms cannot deal with continuous and/or multi-dimensional action spaces). More generally, the lack of a standard representation for continuous state problems with no priors on the transition function, makes it difficult

to rigorously compare algorithms; actually, the only common ground for SDP transition functions seems to be Turing-computability[15], making standard comparison more difficult than in e.g. supervised learning.

This lack of principled assessment framework was one of the motivations behind the development of the *OpenDP* Toolbox, allowing the reader to replicate, modify and extend the presented experiments (see section 3.2).

3.4.1 Introduction

In many applications, dynamic optimization without learning would be intractable. Accordingly, many SDP papers report the use of neural networks [Cou, Cou02], adaptive discretization [MM99a], CMAC [Sut96b],[DS97], EM-learning of a sum of gaussians [YIS99], or various forms of local units [And93, KA97, RP04].

The learning algorithms integrated in *OpenDP*, listed in next subsection, will be compared in the SDP framework as fairly as possible: on the same problems (section 3.2.2), in combination with the same optimization algorithm, and with same time budget for parameter tuning. What makes this comparison different from standard regression-oriented Machine Learning comparison is the fact that learning errors and inaccuracies have very specific impacts and far-fetched consequences in the SDP framework, as discussed in section 3.3.2:

- Learning robustness matters in the sense that worst errors (over several learning problems) are more relevant than average errors;

- Along the same lines, the appropriate loss function (L^2 norm, L^p norm, among others) that should be used during learning is yet to be determined even from a theoretical perspective (the interested reader is referred to [Mun05] for a comprehensive discussion of these points);

- The existence of (false) local minima in the learned function values will mislead the optimization algorithms;

[15] Note that some authors only consider simulation logs.

- The decay of contrasts through time is an important issue, as the function value involves the value expectation in further time steps. Accordingly, the value estimate should preserve the ranking of the states/actions, as well as their associated values; otherwise, small errors in every learning problem, will cause huge deviations in the overall control strategy.

3.4.2 Algorithms used in the comparison

The regression algorithms integrated in *OpenDP* involve the regression algorithms from Weka [WF05] (using the default parameters), the Torch library [CB01] and some open-source algorithms available on the Web. Every regression dataset $\mathcal{E} = \{(x_i, y_i), x_i \in \mathbb{R}^d, y_i \in \mathbb{R}, i = 1 \dots n\}$ is normalized ($mean = 0$, $\sigma = 1$) before learning. The methods implemented in Weka are named with a dot "." (e.g. "rules.ConjunctiveRule", "rules" being the name of the package).

- **lazy.IBk** : k-nearest neighbors algorithm, with $k = 5$. The K^* ([CT95]) algorithm uses the same underlying assumption of instance-based classifiers: similar instances will have similar classes. K^* has been used in preliminary experiments, but results are not reported here as it is always equal or worse than IBk algorithm, but prohibitively slow.

- **functions.RBFNetwork** : implements a normalized Gaussian radial basis function network. It uses the k-means clustering algorithm to provide the basis functions and learns a linear regression (numeric class problems) on top of that. Symmetric multivariate Gaussians are fit to the data from each cluster. It standardizes all numeric attributes to zero mean and unit variance.

- **rules.ConjunctiveRule** : this class implements a single conjunctive rule learner. A rule consists of a conjunction of antecedents and the consequent (class value) for the regression. In this case, the consequent is the distribution of the available classes (or numeric value) in the dataset. If the test instance is not covered by this rule, then it is predicted using the default class distributions/value of the data not covered by the rule in the training data. This learner selects an antecedent by computing the Information

Gain of each antecedent and prunes the generated rule using Reduced Error Pruning (REP). For regression, the Information is the weighted average of the mean-squared errors of both the data covered and not covered by the rule. In pruning, the weighted average of the mean-squared errors of the pruning data is used for regression.

- **rules.DecisionTable** : building and using a simple decision table majority classifier. For more information see [Koh95].

- **trees.DecisionStump** : decision stump algorithm, i.e. one-node decision trees (with one-variable-comparison-to-real in the node).

- **meta.AdditiveRegression (AR)**: Meta classifier that enhances the performance of a regression base classifier. Each iteration fits a model to the residuals left by the classifier on the previous iteration. The final predictor is the sum of the classifiers; smoothing proceeds by varying the learning rate parameter. The base classifier is a decision stump.

- **trees.REPTree** : fast decision tree learner. Builds a regression tree using variance reduction and prunes it using reduced-error pruning (with backfitting). Only sorts values for numeric attributes once.

- **MLP** MultilayerPerceptron (implementation of Torch library), using 10 hidden neurons, 0.001 as stopping criterion, 0.01 learning rate and 0.001 learning rate decay. As for the SVM hyper-parameters for "SVMGauss" and "SVMLap" this setting has been determined from independent regression problems.

- **SVMGauss** Support Vector Machine with Gaussian kernel (implementation of Torch library [CB01]), where the hyperparameters are fixed after heuristic rules determined on independent regression problems[16];

- **SVMLap** Support Vector Machine with Laplacian kernel (implementation of Torch library) and same parametrization as above;

[16] Letting n be the number of examples and d the dimension of the instance space, regularization parameter C is set to $2\sqrt{(}n)$, kernel parameter $\gamma = \frac{1}{10} n^{\frac{1}{d}}$, and insensitivity of the loss function $\varepsilon = 0.1$.

- **SVMGaussHP** Support Vector Machine with Gaussian kernel as in **SVMGauss**, where hyperparameters have been optimized after 5-fold cross validation[17].
- **functions.LeastMedSq** : implements a least median squared linear regression using the robust regression and outlier regression ([RL87]).
- **functions.LinearRegression** : linear regression based on the Akaike criterion [Aka70] for model selection.
- **trees.lmt.LogisticBase** : LogitBoost algorithm (see [OS05]).
- **functions.SimpleLinearRegression** (SLR) : a simple linear regression model.
- **LRK** Kernelized linear regression, involving polynomial or Gaussian kernels to re-describe every example[18] (x,y), and computing the MSE linear regressor in the kernelized space;

3.4.3 Results

As already said, the problems in the benchmark suite are not trivial; for most problems, rewards are delayed until reaching the goal (e.g. for robotics problem); and all problems defeat greedy optimization (e.g., in the stock management problems, the greedy policy is worse than a random one, uniformly selecting a decision − which is also the case in many real-world stock management problems).

The experimental setting is as follows:

- Each learning algorithm is provided 300 examples at each time step (complementary experiments with 500 examples give similar results). The limited number of examples is meant to assess the algorithm performances in a realistic bounded resource context (e.g. when the transition function is computed using a computationally heavy simulator);

[17] The optimization algorithm is the home-made Evolutionary Algorithm "EA" described in Appendix B.1, named EA, using the 5-CV error as fitness function, with 50 fitness computations allowed. Empirically, this optimization of the SVM hyper-parameters was found to be significantly more efficient than standard grid-based search.

[18] Letting $k(\cdot,\cdot)$ denote the kernel defined from $X \times X$ to $\mathbb{R}$, kernelization associates to each instance x in X the n-dimensional real valued vector defined as $((k(x_1,x),\ldots,k(x_n,x))$.

- Examples are sampled using quasi random methods (QR, section 3.5), except for Stock Management problems where examples are sampled using Greedy Low Dispersion methods (GLD, section 3.5) because without GLD no learner achieve a better performance than the random controller;

- The optimization algorithm used together with every learning algorithm is an EA (section 3.3) with 70 fitness evaluations.

Table 3.3 summarizes the results of the experiments; detailed results for every problem and every algorithm will be found in Appendix B.3, including the computational cost[19].

All the detailed results on each problem for each learner can be found on Appendix B.3. The table 3.3 summaries and synthesizes the results of learning methods in all problems, for the "300 learning points" experiments. The experiments using 500 give essentially the same results.

We only report the results for the seven best methods and for the best among "greedy" and "stupid". "All bad" denotes when no learning algorithm beats both the "greedy" and "stupid" algorithms. All local optimizations have been performed by the naive genetic algorithm provided in the source code, allowed to use 70 function evaluations.

Support Vector Machine algorithms gave the best results over a large number of problems and got by far the most stable results, which is consistent with their general good behavior in supervised learning. The computational cost was very limited in the range of the experiments, due to the few examples available. Still, the overall computational complexity should scale up quadratically (i.e. poorly) with the number of examples. Firstly, the SVM complexity is quadratic in the number of training examples. More importantly, the model learned has linear complexity wrt the number of support vectors, and in practice for regression problems, linear complexity with the number of training examples too. Letting respectively m, N, n and T denote the allowed number of function evaluations in the optimizer, the number of transitions used to compute the value expectation in the Bellman equation, the number of sampled states and the horizon, it comes that the whole dynamic programming procedure requires $m \times N \times n \times T$ calls to the learned model; denoting αn the

[19]While simulation costs are negligible in the benchmark problems for obvious reasons, in real-world problems the learning cost can be a small fraction of the overall computational cost.

Problem	Dim.	Best algo.	SVM Gauss/Lap in best group	SVMGaussHP worse than SVMGauss
Stock and Demand	4	SVM	y	y
	8	IBk	n	y
	12	All bad	N/A	N/A
	16	All bad	N/A	N/A
Stock and Demand V2	2	SVM	y	n
	4	SimpleLinearR	n	n
Fast Obst. Avoid.	1	IBk	n	y
Many Obst. Avoid.**	1	LRK(poly)	n	y
Many bots**	4	SVM	y	n
	8	SVM	y	y
	12	SVM	y	y
	16	SVM	y	y
Arm*	3	SVM	y	n
	6	SVM	y	n
	9	LeastMedSq	y	n
	12	LeastMedSq	y	n
Away*	2	IBk	y	y
	4	SVM	y	n
	6	SVM	y	n
	8	SVM	y	n
Total			14/18	8/18

Table 3.3: Comparative results of the learning algorithms in the SDP framework. Column 3 indicates the best algorithm, which is SVM on 11 out of 18 problems (SVM include both SVMGauss and SVMLap, as their performances are never statistically different). Column 4 reads y if SVM algorithms are not significantly different from the best algorithm, which is the case on 14 out of 18 problems; in all cases, they are never far from the best one (and significantly the best in 11/18). Column 5 indicates whether SVM-HP is significantly worse than SVM, which is the case for 8 out of 18 problems (see text for comments).

number of support vectors, with $\alpha < 1$, the procedure complexity is thus quadratic in the number of training examples.

Interestingly, SVM-HP (where the SVM hyper-parameters are optimized using 5fold CV) is dominated by standard SVM (where hyper-parameters are set using simple heuristic rules). Not only is SVM-HP more computationally expensive; also, its overall performance in the ADP framework is often significantly worse. While it might seem counter-intuitive

(tuning the hyper-parameters by CV is considered to be the best practice wrt the standard minimization of the mean square error in a supervised learning context), this result is consistent with the analysis presented in section 3.3.2, discussing the robustness requirements in the dynamic programming context. Indeed, the worst learning error over several learning problems matters more than the average learning error. The use of the heuristic rules thus appears to be more conservative and more capable to prevent catastrophic failures than a more sophisticated tuning procedure. Accordingly, further research might consider to use a more conservative criterion for the optimization of hyper-parameters, e.g. based on the maximum error over all folds as opposed to the average error.

The popular Multilayer Perceptron algorithm is often the second best one, with performances just below that of SVMs. It is also very fast, with linear learning complexity in the number of examples (the number of hidden neurons is fixed); and the complexity of the model learned is constant. This suggests that MLP might catch up SVM, at least from a computational viewpoint, when dealing with larger numbers of examples (and if the DP procedure does spend much time in learning and calling the model learned). The MLP performances might be improved by tuning its hyper-parameters[20]; on the other hand, the MLP performances seem to be adversely affected by the problem dimension in the range of experiments (Appendix B.3).

In conclusion, these experiments suggest that SVM and MLP learning algorithms are well suited to the SDP framework, although much remains to be done in order to be able to recommend a given learning algorithm for a given RL problem.

3.5 Sampling in SDP

This section is devoted to the sampling task involved in the estimation of the Bellman value function. As already mentioned (section 3.1.3), the ability of the learner to select examples and modify its environment in order to get better examples, referred to as active learning, is one of the key sources of efficiency for learning systems [CGJ95b]. Two main types of active learning are distinguished: *blind approaches* only consider the instance

[20]Still, it was found more difficult to tune or provide heuristic setting for the MLP hyperparameters than for the SVM ones, in terms of manual labor.

distribution [CM04]; *non-blind approaches* also exploit the label of instances previously selected, mediated through (an ensemble of) hypotheses learned from the labelled instances [LG94, SOS92, SC00, CGJ95a].

After discussing the state of the art, this section investigates two main issues. The first one regards the comparative performances of blind and non-blind approaches for active regression learning. Experimental results are provided as well as a theoretical analysis of randomized sampling. The second issue regards blind approaches and the appropriate "degree of randomness" to be involved in blind samplers. A random-light blind sampler is described; theoretical results establishing its universal consistency, together with improved convergence rate in the case of "sufficiently smooth" learning problems, are given.

3.5.1 State of the art

Despite the fact that non-blind approaches are provided with more information, their superiority on blind approaches is unclear in most cases, due to the excellent robustness of blind approaches. Actually, avoiding the domain regions which can be considered as uninformative on the basis of the current hypotheses, might lead to miss relevant regions. From a theoretical perspective, [Vid97b] establishes negative results about active learning (worst-case analysis). From an experimental standpoint, the successes of active learning seem to concern limited hypothesis spaces (e.g. decision trees more than neural networks or SVM) and limited target models (classification more than regression problems). Along these lines, the success of viability[21] approaches [CD06] in Reinforcement Learning might be due to the fact that it actually tackles a classification problem (entailing some loss of generality). The greater difficulty of active regression, compared to classification or learning regression trees, can be viewed as it defeats simple heuristics such as selecting instances close to the domain boundary.

A second issue concerns randomized versus deterministic sampling approaches. As already mentioned, the traditional application of quasi-random sampling is numerical integration. In this context, deterministic blind samples significantly improve on random

[21] A viability problem considers only the "survival" of the agent rather than a precise performance, e.g. wether the agent reaches the goal or not rather the actual time to reach the goal.

samples (wrt to various performance criteria) in small dimension; but when the dimension increases, strictly deterministic approaches [SW98] show strong limitations (see also [KMN99, Rus97] for some interesting properties of random sampling in the specific case of control problems). Current trends in the domain advocate the use of *randomized quasi-random* sequences, in particular in medium and large dimensions [LL02].

Lastly, active learning presents some specificities in the RL context. On the one hand, RL intrinsically is an Exploration vs Exploitation problem, where exploitation (learning) is tightly intertwined with exploration (gathering information); on the other hand, RL raises active regression problems whereas the literature mostly considers active classification problems. Finally, many works related to active sampling in RL proceed by actively driving the environment simulator; see e.g. the pioneering work [BBS93] and many followers. While these approaches have strong advantages, the simulator can also miss interesting regions in the state space due to poor initial policies. For this reason, the presented study will focus on the efficient sampling of the whole domain; early works along this line are [MM99b] (active discretization of the domain) and [CD06] (active SVM learning in a viability framework, casting the regression task as a classification task).

The rest of the section will present our contributions, with the goal of achieving some efficient tradeoff from both theoretical and algorithmic viewpoints, between i) blind and non-blind approaches, able to take advantage of prior knowledge about the sampling; ii) randomized and deterministic samplings.

3.5.2 Derandomized sampling and Learning

After introducing formal background in derandomized sequences, we shall study the robustness properties of a random-light sampler.

In the following, E^* denote the set of finite sequences in set E.

Definition 3.5.1 *Let domain $D = [0,1]^d$. A* **learner** *A on D is a computable mapping from $(D \times \mathbb{R})^*$ to $\mathbb{R}^D$. Let $\mathcal{A}$ the set of learners. A* **sampler** *(respectively, a blind sampler) S on D is a computable mapping from $(D \times \mathbb{R})^* \times \mathcal{A}$ (resp, from $(D)^*$) to D. An* **active-learner** *$S+A$ on D, made of sampler S and learner A, is an algorithm parametrized from a real-valued target function f on D, with pseudo-code:*

1. *For* $n = 0;;n++$
2. *sample* x_n *in D,* $x_n = S((x_0,y_0)\ldots,(x_{n-1},y_{n-1}),A)$*;*
3. *call* $y_n = f(x_n)$*;*
4. *learn* $f_n = A((x_0,y_0),\ldots,(x_n,y_n))$
5. *endfor*

At each iteration, $S+A$ *outputs current hypothesis* f_n*.*

Sampler S is said to be **almost-deterministic** *(AD) if there exists a bounded function k such that S only uses* $k(n)$ *random bits to select* $x_1,\ldots,x_n$*. Learner A is said to be* **almost-deterministic** *(AD) if for each run it only uses a finite number of random bits that only depends on the length of its inputs. Active learner* $A+S$ *on D is said to be* **universally consistent** *(UC) if, for any measurable f with values in* $[0,1]$*, the* L_2 *norm of the difference* $||f-f_n||_2$ *goes to 0 almost surely as* $n\to\infty$*. Sampler S on D is said to be* **universally consistent** *if there exists at least one almost-deterministic learner A, such that* $S+A$ *is universally consistent.*

The definition of UC samplers refers to almost deterministic learners, in order to make a clear distinction between AD and randomized samplers (otherwise, the learner stochasticity could be used in order to create some stochasticity in the sample points).

Theorem 3.5.2 *Random sampling is universally consistent.*

Proof: Follows from the universal consistency of learning algorithms in the statistical learning literature (see e.g. [DGKL94]).

Theorem 3.5.3 (UC samplers are stochastic) *Let S be an AD-sampler. Then, for any AD-learner A,* $S+A$ *is not UC. Therefore, S is not UC.*

Proof: Let S and A respectively denote an AD-sampler and an AD-learner. Then, consider $x_0,\ldots,x_n,\ldots$ the (possibly stochastic) sequence of points provided by S if f is the target function identically equal to 1.

By definition of AD, x_i ranges in a enumerable domain V. Let g_p, for $0 \leq p \leq 1$, denote the function equal to 1 on V, and to p elsewhere. Assuming $S+A$ is UC, then its output f_n should a.s. converge toward f in norm L^2 as n goes to infinity; however, f_n only depends on the sequence $x_0, \ldots, x_n, \cdots \in V$; therefore f_n should also converge toward g_p whatever the value of p is, which shows the contradiction. □

Theorems 3.5.2 and 3.5.3 establishes that random sampling is universally consistent while almost deterministic sampling cannot be universally consistent. The question thus is the amount of randomness required to enforce UC. A moderately randomized sampler with UC property is introduced below. This sampler is based on random shift of quasi-random sequences, the good convergence rate (toward smooth target function) of which has been proved by [CM04].

Let us first establish a Lemma about uniform low-dispersion sequences.

Definition 3.5.4 *Let $P = \{x_1, \ldots, x_n, \ldots\}$ be a sequence of points in $D = [0,1]^d$. For each point x in D, let $NN_n^P(x)$ denote the nearest neighbor of x (ties are broken randomly) in the n^{th} first points of sequence P: $x_1, \ldots x_n$. The dispersion $Dis_n(P)$ is the maximum over x in D, of the L_∞ distance between x and $NN_n^P(x)$.*

$$Dis_n(P) = \sup_{x \in D} \inf_{i=1\ldots n} ||x - x_i||_\infty$$

Let E denote a subset of D; $\hat{P}_n(E)$ denotes the fraction of $x_i, i \leq n$ that belong to E.

$$\hat{P}_n(E) = \frac{|\{i \; s.t. \; x_i \in E, i \leq n\}|}{n}$$

Let x a point in D, let $R(x)$ denote the axis-parallel hyperectangle in D with $(0, \ldots, 0)$ and x as vertices, and denote $\hat{P}_n(R(x))$ the fraction of points $x_i, i \leq n$ that belong to $R(x)$. Let μ the Lebesgue-measure.

The discrepancy of P is the maximum over x in D, of the difference between $\hat{P}_n(R(x))$ and $\mu(R(x))$.

Lemma 3.5.5 (Properties of well-distributed shifted sequences) *Let $P = x_1, \ldots, x_n, \ldots$ be a sequence in D with dispersion decreasing like $O(n^{-1/d})$ and discrepancy decreasing*

to 0 as n goes to infinity:

$$Dis_n(P) = O(n^{-1/d}) \tag{3.5}$$

$$\lim_{n\to\infty} \sup_{x\in D} |\hat{P}_n(R(x)) - \mu(R(x))| = 0 \tag{3.6}$$

Let s be a random variable uniformly distributed in D. Let sequence $Q = x'_0, \ldots x'_n, \ldots$ *be defined from P with* $x'_i = x_i + s$ *modulo* 1 *(i.e.* x'_i *is such that* $x_i - x'_i + s \in \mathbb{Z}^d$ *and* $x'_i \in D$*).*

Let $\mathcal{B}_n^{(i)}$ *denote the set of points x with* $NN_n^Q(x) = x'_i$*. Then:*
(i) For some $\zeta(n) = O(1/n)$ *as* $n \to \infty$*,*

$$\forall n \in \mathbb{N}, \forall i \leq n, \mu(\mathcal{B}_n^{(i)}) \leq \zeta(n) \tag{3.7}$$

and for any measurable E,

$$\text{almost surely in s, } \hat{Q}_n(E) \to \mu(E) \tag{3.8}$$

(ii)

$$\forall \delta > 0,\ \exists N(\delta),\ \forall n \geq N(\delta),\ \sup_{x\in[0,1]^d} \inf_{i\leq n} ||x - x'_i|| \leq \delta \tag{3.9}$$

Proof: Equations 3.5 and 3.6 hold for sequence Q as they are invariant under translation (modulo 1):

$$Dis_n(Q) = \sup_{x\in D} \inf_{i=1\ldots n} ||x - x'_i||_\infty = O(n^{-1/d}) \tag{3.10}$$

$$\lim_{n\to\infty} \sup_{x\in D} |\hat{Q}_n(R(x)) - \mu(R(x))| = 0 \tag{3.11}$$

Eq. 3.7 and 3.9 follows from eq. 3.10.

Lastly, eq. 3.8 is obtained as follows. On the one hand, a well-known result in the discrepancy literature is that eq. 3.11 entails that $\hat{Q}_n(E)$ goes to $\mu(E)$ for every axis-parallel rectangle E.
Let E now denote a Lebesgue-measurable set. For any $\varepsilon > 0$, there exists two finite sets of

axis-parallel hyperectangles denoted $R_i, i \in I$ and $R'_j, j \in J$ such that

$$D \setminus \bigcup_{i \in I} R_i \subset E \subset D \setminus \bigcup_{j \in J} R'_j \tag{3.12}$$

$$\mu\left(D \setminus \bigcup_{j \in J} R'_j \setminus (D \setminus \bigcup_{i \in I} R_i) \right) \leq \varepsilon \tag{3.13}$$

The above equations ensure that $|\hat{Q}_n(E) - \mu(E)|$ is less than 2ε for n sufficiently large. $\square$

Theorem 3.5.6 (UC with random shift) *Let sampler S be defined as follows:*

1. Randomly uniformly draw $s \in D = [0,1]^d$.

2. Let $P = x_0, \ldots, x_n, \ldots$ be a deterministic sequence in D with dispersion $Dis_n(P) = O(n^{-1/d})$ and bounded discrepancy

$$\sup_{x \in D} |\hat{P}_n(R(x)) - \mu(R(x)) = \underbrace{o(1)}_{n \to \infty}$$

3. Let S output the sequence Q defined as $x'_n = x_n + s$ modulo 1 (i.e. x'_n is such that $x_n - x'_n + s \in \mathbb{Z}^d$ and $x'_n \in [0,1[^d$).

Then, S is UC.

Interpretation: While theorems 3.5.3 and 3.5.6 show that randomness is required for the UC property, the above theorem shows that randomly shifting a deterministic sequence is enough to get a UC sampler, with same improved convergence rates as in [CM04] for smooth target functions.

Proof: Let A be the 1-nearest neighbor classifier for the L^∞ distance; we shall show that $S + A$ is UC, thus establishing that S is UC. Let $T_n(h)$ denote the function $x \mapsto h(NN_n^Q(x))$, i.e. T_n is the active learner $S + A$ using n examples.

After Lemma 3.5.5 and with same notations, for any $\delta > 0$ there exists $N(\delta)$ such that for all $n \geq N(\delta)$

$$sup_{x \in [0,1]^d} ||NN_n^Q(x) - x|| \leq \delta \tag{3.14}$$

and also that for some $\zeta(n) = O(1/n)$,

$$\forall n \in \mathbb{N}, \sup_{i=1\ldots n} \mu(\mathcal{B}_n^{(i)}) \leq \zeta(n) \tag{3.15}$$

and for any measurable E, almost surely in s,

$$\hat{Q}_n(E) \to \mu(E) \tag{3.16}$$

- After the theorem of Lusin, for any f measurable from $[0,1]^d$ to $[0,1]$, there exists a sequence f_n of continuous functions from $[0,1]^d$ to $[0,1]$ that is equal to f expect on a measureable set of measure decreasing to 0. Hence, for each $\varepsilon > 0$, there exists $n(\varepsilon)$ such that
$$n \geq n(\varepsilon) \Rightarrow ||f_n - f||_1 \leq \varepsilon$$

- Every f_n, being a continuous function defined on compact $[0,1]^d$, is uniformly continuous. Let $\delta_{\varepsilon,n}$ be such that
$$||x-y||_\infty \leq \delta_{\varepsilon,n} \;\Rightarrow\; |f_n(x) - f_n(y)| \leq \varepsilon$$

- For $n \geq n(\varepsilon)$ and $n' \geq N(\delta_{\varepsilon,n})$, it follows from eq. 3.14:
$$||f_n - f||_1 \leq \varepsilon \tag{3.17}$$
$$||T_{n'}(f_n) - f_n||_1 \leq ||T_{n'}(f_n) - f_n||_\infty \leq \varepsilon \tag{3.18}$$

- Combining equations 3.17 and 3.18 above, it comes:
$$||f - T_{n'}(f_n)||_1 \leq 2\varepsilon \tag{3.19}$$

- Let E_A denote $\{x; |f(x) - f_n(x)| \geq A\}$; after eq. 3.17,
$$A\mu(E_A) \leq \int_{E_A} |f_n(x) - f(x)| \leq \int |f_n(x) - f(x)| \leq \varepsilon$$
$$\mu(E_A) \leq \varepsilon/A \tag{3.20}$$

- Applying eq. (3.16) with $E = E_A$, with eq. (3.20), leads to:

$$\hat{Q}_{n'}(E_A) \leq \mu(E_A) + o(1) \leq \frac{\varepsilon}{A} + \underbrace{o(1)}_{n' \to \infty} \tag{3.21}$$

  almost surely in s.

- The two functions $T_{n'}(f)$ and f are equal on the set $\{x'_1, \ldots x'_{n'}\}$; therefore, with $E_1 = \{i \in 1..n'; |T_{n'}(f)(x'_i) - f_n(x'_i)| \geq A\}$, equation 3.21 yields to

$$\frac{1}{n'} \# E_1 \leq \varepsilon / A + \underbrace{o(1)}_{n' \to \infty} \tag{3.22}$$

- eqs (3.22) and (3.15), with $E_2 = \{x \in D; |T_{n'}(f)(x) - T_{n'}(f_n)(x)| \geq A\}$, lead to

$$\mu(E_2) \leq (\varepsilon / A + o(1)) \underbrace{n' \zeta(n')}_{O(1)}$$

  which in the case $A = \sqrt{\varepsilon}$ leads to

$$||T_{n'}(f) - T_{n'}(f_n)||_1 \leq O\left(2\sqrt{\varepsilon} + \underbrace{o(1)}_{n' \to \infty}\right) \tag{3.23}$$

- Eqs 3.19 and 3.23 lead to

$$||T_{n'}(f) - f||_1 \leq O(\varepsilon) + O(\sqrt{\varepsilon}) + \underbrace{o(1)}_{n' \to \infty} = O(\sqrt{\varepsilon})$$

  almost surely in s, for any fixed ε. In particular, this is true almost surely for all $\varepsilon = 1/2^i$ for $i \in \mathbb{N}$ (as this set is countable), entailing the almost sure convergence of $T_{n'}(f)$ to f for the L_1 norm and therefore for the L_p norm for any $p \geq 1$. □

Remark: This theoretical analysis only requires the discrepancy to decrease to 0 (while so-called low-discrepancy sequences assume the discrepancy to decrease like $O(\log(n)^d/n)$. Its scope might appear to be limited as in practice all algorithms are almost

deterministic ones (the number of random bits being used is limited). However, beyond the theoretical interest, it shows that the best of both worlds (UC, fast convergence properties in small dimensions) can be reached by a moderate stochasticity (random shift) on the top of a deterministic sequence. Likewise, current approaches in low-discrepancy sequences similarly combines some randomness (for robustness in high dimensions) with deterministic sequences (for the fast convergence properties).

3.5.3 Sampling methods (SM) and applications

This subsection discusses the role of the sampler in the SDP framework and describes the scope of the experimental study.

The role of the sampler is to reduce the computational cost − a main issue of stochastic dynamic programming − while preserving its accuracy.

Specific samplers have been developed for dynamic problems. [BBS93] showed the importance of ”reducing” the domain, when possible, by using simulations to focus the search in the a priori relevant regions of the full state space. The drawback is that some prior knowledge, e.g. some approximate solution, is needed to apply simulations; while simulations are needed to further focus the search and apply dynamic programming. The robustness of the approach thus strongly depends on the quality of the priors. The main advantage is its scalability, as it can deal with much higher dimension domains than the approach sampling the whole search space. [Thr92] studied how to avoid visiting many times the same area (leading to better convergence rates in some theoretical framework), and thus reducing the curse of dimensionality.

In the following, only general-purpose samplers will be considered, as our goal is to assess the “out-of-shelf” performance of algorithms without assuming expert knowledge. These samplers are described in next two subsections.

3.5.4 Blind Samplers

Let us describe blind samplers based on low-dispersion and low-discrepancy sequences, referring the interested reader to [Tuf96, Owe03, LL02] for a general introduction to ”well chosen” sets of points.

Consider $P = \{x_1, \ldots x_n\}$ a sequence in $D = [0,1]^d$.

Low discrepancy. Discrepancy is most usually defined as the maximum over $x \in D$, of the difference $|\hat{P}_n(R(x)) - \mu(R(x))|$. For iid uniformly distributed P, discrepancy roughly decreases as $O(1/\sqrt{n})$. Well chosen deterministic or (non-naive) randomized points achieve $O(\log(n)^d/n)$. Exponent d thus severely affects the discrepancy in high dimensions ($d > 5$). Many solutions have been proposed for dealing with high dimension domains, based on some prior knowledge about the structure of the search space (with strong dependencies among the coordinates [Hic98], or assuming some clustering on the search space [Owe03]). In the general case on $D = [0,1]^d$, we are only aware of the following two results (see [WW97, SW98] and references therein):

- the existence of deterministic sequences with discrepancy decreasing $O(1/(n)^\alpha)$ with $\alpha < \frac{1}{2}$, i.e. the sequence is worst than a uniform iid sequence, with a constructive proof.
- the existence of deterministic sequences with discrepancy decreasing $O(1/(n)^\alpha)$ with $\alpha > \frac{1}{2}$, i.e. the sequence is better than random and better than standard low-discrepancy sequences in high dimensions. Unfortunately, the proof is not constructive; we are not able to construct such a sequence.

Other definitions (see [Nie92, Tuf96, Owe03]) involve the use of general polygons instead of axis-parallel rectangles, or use a L_p norm to measure the loss $|\hat{P}_n(R(x)) - \mu(R(x))|$.

Direct optimization of the discrepancy has been attempted [Auz04] and found to be very difficult due to the multi-modality of the landscape. Mainstream approaches are based on algebraic procedures. In the following, we shall use standard Niedereiter sequences [Nie92], referred to as "low-discrepancy sequence" (quasi-random, QR).

Low dispersion. As mentioned in section 3.3.3, the low dispersion criterion is less widely used than low-discrepancy, although it has some advantages (see e.g. discussions in [LL03, LBL04]). The most usual criterion (to be minimized) is, with d the Euclidean distance:

$$Dispersion(P) = \sup_{x \in D} \inf_{p \in P} d(x,p) \tag{3.24}$$

It is related to the following one, to be maximized, which defines an easier optimization problem (except when n becomes large):

$$Dispersion_2(P) = \inf_{(x_1,x_2)\in P^2} d(x_1,x_2) \tag{3.25}$$

However, the above criterion is biased toward the points on the frontier ∂D of the domain; to avoid this bias, criterion $Dispersion_3$ is defined as:

$$Dispersion_3(P) = \inf_{(x_1,x_2)\in P^2} d(x_1,\{x_2\}\cup\partial D) \tag{3.26}$$

In the following, **low-dispersion point set** (LD) is referred to as a finite sequence optimizing eq. 3.25.

A **greedy-low-dispersion point set** (GLD) is a point set constructed after the greedy iterative optimization algorithm[22], defining starting point $x_1 = (0.5,...,0.5)$ as the center of $D = [0,1]^d$, and x_n such that it optimizes $Dispersion_2(\{x_1,\dots x_n\})$ conditionally to $\{x_1,\dots,x_{n-1}\}$. See Fig. 3.1 for an illustration.

A **greedy-low-dispersion point set far-from-frontier** (GLDfff) is based on the same greedy optimization and initialization, optimizing eq. 3.26. The use of other L_p distances instead of the Euclidean one did not lead anywhere.

3.5.5 Non-blind Sampler

While many approaches have been developed for active learning (see Chapter 1), they are not appropriate to SDP as most of them address classification problems [Ton01] and existing active regression algorithms are moderately efficient.

Therefore a specific approach, based on *Evolutionary algorithms* and inspired from [LW01], has been designed and implemented in *OpenDP*. To our knowledge, this is a new approach for active learning in SDP, with three main advantages: i) its baseline is random sampling (the initial population); ii) it is anytime (working under bounded resources); iii) it allows one to explore for free many criteria.

[22]The implementation is based on Kd-trees; the extension to Bkd-trees (allowing for fast online adding of new points, [PAAV02]) is possible.

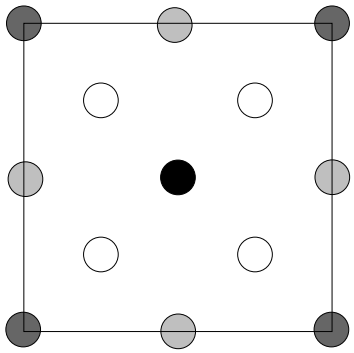

Figure 3.1: A GLD-sample in dimension 2. Ties are broken at random, resulting in a better distribution of the points on average. Dark points are those that are selected first; the selection among the grey points is random.

As introduced in section 3.3, *Evolutionary algorithms* are population-based optimization algorithms, targeted at non-differentiable and non-convex fitness functions; the interested reader is referred to [Bäc95, ES03] for a comprehensive introduction. *Evolutionary algorithms*-based samplers proceed as follows: 1. generate an initial population uniformly on the domain; 2. evolve the population until the allowed number of fitness evaluations; 3. use as active sample the union of all populations in all generations. Note that the sampler does not embed any learning algorithm.

We used the home-made *Evolutionary algorithms* mentioned in section 3.3.3 and Appendix[23] B.1. Population size is $n = N^{\alpha}$, where $\alpha \in]0,1]$ is a parameter of the approach and N is the number of sampled points. The fitness function is the cost function, to be minimized. The resources are bounded from the maximal number of fitness evaluations.

Actually, the non-blind sampler proposed is biased toward sampling the small value region of the target function V (or rather, from its current approximation V_n), based on the fact that the small value region is the most interesting one. (the target function V, sum of the instantaneous cost and the expected cost-to-go, must be minimized). If the optimal trajectories are close the small value regions, this bias is very efficient, while it may lead to poor results on some other problems such as stock management.

[23] We used the default parameters $\sigma = 0.08, \lambda_1 = 1/10, \lambda_2 = 2/10, \lambda_3 = 3/10, \lambda_4 = 4/10$.

3.5.6 Experiments

This section reports on the experiments done to assess various samplers in the SDP framework. All experiments are reproducible from the *OpenDP* Toolbox.

Experimental setting

Each sampler produces 500 points and is assessed in combination with the same learning algorithm, selected after section 3.4, that is SVM with Gaussian kernel (using the SVM-Torch implementation [CB01], integrated within *OpenDP*), with rule-based setting of the hyper-parameters (section 3.4).

Each sampler produces a 500-points sample in each learning step. The performance of the sampler is the cost of the best strategy (to be minimized), averaged on 44 independent runs with same parameters. The benchmark problems (section 3.2.2) are integrated in the *OpenDP* Toolbox.

The search space $S \times M$ involves the continuous state space $S = [0,1]^d$ and the finite space $M = \{m_1, \ldots, m_k\}$ describing the exogenous Markov process (sections 1.1.4 and 3.2.2). All assessed samplers are integrated in the *OpenDP* Toolbox:

1. Sampling modules (SM) GLD, QR, LD, GLDfff are the blind samplers defined in section 3.5.4. Each module first extracts a N sample $s_1 \ldots, s_N$ on S, and returns the $N \times k$ sample defined as (s_i, m_j), for $i = 1 \ldots N, j = 1 \ldots k$.

2. Random samplers include SM RandomForAll (uniform iid on $S \times M$), SM RandomOnlyForContinuousStates (ROFCS) ((s_i, m_i) is defined as s_i is uniform iid on S and each element in M occurs an equal number of times). SM DeterministicForDim1 only differs from ROFCS as the first coordinate in s_i is drawn on the same regular grid for each m_j (other coordinates being iid on $[0,1]$; it thus returns a deterministic if S is one-dimensional ($d = 1$).

3. EAS-α is the non-blind sampler described in section 3.5.5, with parameter α (α control both the population size and the number of generations).

Experimental Results of Blind samplers

The detailed performances of all blind samplers are reported in Tables 3.5 and 3.6, and summarized in table 3.4:
Overall, the best sampler in the range of experiments is the Greedy Low Dispersion (GLD) one; it significantly outperforms baseline random samplers (ROFCS and Random) on 6 out of 8 problems. When relevant ($k > 1$), the exhaustive exploration (derandomized discrete sampler) of the discrete M space is critical to the overall performances. With poor discrete samplers (Random and RandomForAll), the results are poor; a exhaustive discrete sampler combined with basic continuous sampler (ROFCS) catches up more sophisticated mixed samplers (Quasi-Random and GLDfff). More precisely, Quasi-Random and GLDfff significantly outperform ROFCS on only, respectively 4 and 3 problems out of 8. Finally, ROFCS is the best sampler on 2 out of 8 problems, the ManyBots (dimension 8) and the Arm (dimension 3) problems.

Column 1 indicates the discrete and continuous dimensions k and d of the problem. Column 2 indicates whether GLD significantly outperforms all other samplers (legend **y**) or all samplers except the second best (*y*(= second best)). Columns QR and GLDfff respectively indicate whether QR (resp. GLDfff) significantly outperforms ROFCS (legend y). More detailed results are provided in tables 3.5 and 3.6.

Experimental Results of Non-blind samplers.

Along the same experimental setting (500 points in each sample, average results on 44 runs), the *Evolutionary algorithms*-based sampler was experimented and compared with randomized and derandomized blind samplers on all benchmark problems. Overall, *EA-sampler* appears to be poorly competitive with other algorithms in small and medium dimension ($d < 8$). In higher dimensions, GLD outperforms *EA-sampler* on Fast- and Many-obstacle avoidance (dim 8) while *EA-sampler* outperforms GLD on Arm (dim 12) and Many-bots (dim 8), whatever the α value is. For Away (dim 8), *EA-sampler* outperforms GLD significantly for $\alpha < .95$.

Overall, *EA-sampler* seems to be competent in higher dimensions. For some problems, it significantly outperforms all other algorithms, which is interpreted as *EA-sampler* rapidly

Problem (dim: continuous+discrete)	GLD 1st ranked	QR	GLD	GLDfff
Stock (4+1)	**y**	n	y	n
Fast Obst. Avoid. (2+0)	y (=GLDfff)	y	y	y
Fast Obst. Avoid. X4 (8+0)	**y**	y	y	y
Many Obst. Avoid. (2+0)	**y**	y	y	n
Many Obst. Avoid. X4 (8+0)	**y**	y	y	n
Many bots. (8+0)	n	n	n	y
Arm (3+2)	n	n	n	n
Away (2+2)	y (=QR)	n	y	n
Total	6/8	4/8	6/8	3/8

Table 3.4: Summary of the results: Derandomized vs Randomized blind samplers. Column 1 indicates the problems with dimensions of the continuous state space and the (discrete) exogenous random process space. Column 2 indicates whether GLD significantly outperforms all other samplers; Column 3 (resp. 4) indicates whether QR (resp. GLDfff) significantly outperforms the randomized sampler. Overall, the best choice in the range of experiments seems to combine the exhaustive exploration of the discrete search space, with derandomized sampler GLD on the continuous search space.

focuses the search on "good regions" (where the expected cost-to-go is small) while the search space includes large bad regions, which can be safely avoided due to the existence of reasonable good paths. The lesser performance of blind randomized and derandomized samplers is explained as these must explore the large bad regions. Meanwhile, *EA-sampler* shows performances similar to that of randomized approaches in "Arm" and "Away" problems.

3.5.7 Discussion

The scope of the study is restricted to active learning problems in the dynamic programming context. The main differences compared to mainstream active learning [CGJ95a] concern the learning target (regression instead of classification) and the learning criteria (robustness and L_∞ error instead of misclassification rate). Both differences might explain why blind samplers consistently dominate non-blind samplers at least in small and medium dimensions.

Specifically, our theoretical study firstly establishes that some randomness is required

Fast Obstacle Avoidance			
Sampler	Score	Std	time (s)
GLD	550	±1.e-16	11.1735
GLDfff	550	±1.e-16	14.8453
QR	913.131	±15.77	11.4533
DFDim1	975.758	±8.59	13.9477
LD	975.758	±9.90	16.5256
Random	976.263	±9.34	11.1441
ROFCS	992.424	±4.99	12.9234

Many Obstacle Avoidance			
Sampler	Score	Std	time (s)
GLD	60	±1.e-16	8.2013
QR	73.3838	±1.06	8.4808
LD	76.6667	±1.22	11.6583
ROFCS	78.1818	±1.34	8.28459
DFDim1	78.3333	±1.23	8.34835
Random	79.2424	±1.34	8.33714
GLDfff	100	±1.e-16	8.30003

Fast Obstacle Avoidance X4			
Sampler	Score	Std	time (s)
GLD	550	±1.e-16	230.85
GLDfff	553.409	±1.92	238.737
QR	553.409	±1.92	234.005
Random	556.818	±3.08	217.76
DFDim1	559.091	±3.36	217.753
ROFCS	563.636	±5.23	217.473

Many Obstacle Avoidance X4			
Sampler	Score	Std	time (s)
GLD	64.5455	±0.21	176.867
QR	83.2955	±2.57	180.293
ROFCS	84.3182	±2.53	172.537
DFDim1	88.8636	±2.41	172.924
Random	90.9091	±2.17	172.471
GLDfff	100	±1.e-16	177.52

Table 3.5: Derandomized and Randomized Samplers on the "Fast Obstacle Avoidance" and "Many-Obstacle Avoidance" problems with baseline dimension d (Left) and dimension $4 \times d$ (Right; LD sampler omitted due to its computational cost). GLD dominates all other algorithms and reaches the optimal solution in 3 out of 4 problems. The comparison between GLD and GLDfff illustrates the importance of sampling the domain frontiers: the frontiers are relevant for problems such as the fast-obstacle-avoidance problem and GLDfff catches up GLD (although all algorithms catch up for dimension $\times 4$). But the domain frontiers are less relevant for the many-obstacle-avoidance problem, and GLDfff gets the worst results. Finally, randomized and derandomized samplers have similar computational costs, strongly advocating the use of derandomized samplers for better performances with no extra-cost.

for a sampler to reach universal consistency (which is missed by deterministic or almost deterministic samplers in worst case analysis, thm 3.5.3)[24]. Secondly, the use of moderate randomization (e.g. random shifting) on the top of a derandomized sequence is sufficient to get the best of both worlds (thm 3.5.6), namely UC [DGKL94] and good convergence rates for smooth target functions [CM04]. This result is experimentally supported by the good performances of the GLD blind sampler (Table 3.4).

Besides the theoretical and experimental aspects, one main contribution of this chapter thus is the new GLD blind sampler, anytime extension of regular grid-sampling in $\mathbb{R}^d$ (see

[24]This result supports the claim that randomness provides robustness [Rus97]. A related result is that quasi-randomized sequences have to be randomized to enforce the absence of domain-specific biases [LL02].

Stock Management			
Sampler	Score	Std	time (s)
GLD	1940.03	±2.03	24.9649
ROFCS	3002.01	±7.58	18.6261
Random	3015.05	±5.23	16.7896
DFDim1	3028.23	±3.02	16.9017
LD	3030.73	±1.74	25.6151
QR	3031.08	±1.64	16.3847
GLDfff	3031.96	±1.56	17.2446

Arm			
Sampler	Score	Std	time (s)
ROFCS	10.898	±0.17	25.1159
Random	10.8981	±0.18	25.2459
QR	13.1177	±0.19	20.3092
GLDfff	13.2963	±0.18	19.0619
LD	13.6147	±0.18	36.1059
DFDim1	13.71	±0.19	21.364
GLD	13.8206	±0.17	20.3117

Many bots			
Sampler	Score	Std	time (s)
GLDfff	200.505	±2.01	13.9471
ROFCS	246.667	±2.70	14.6795
Random	248.081	±2.89	14.6508
LD	249.293	±2.35	18.1841
DFDim1	251.717	±2.91	14.6166
QR	252.525	±3.24	14.6855
GLD	340.101	±4.25	14.3905

Away			
Sampler	Score	Std	time (s)
GLD	1.90404	±0.17	20.4491
ROFCS	2.05556	±0.18	19.6853
QR	2.18687	±0.22	20.1727
DFDim1	2.20707	±0.21	19.6218
LD	2.26263	±0.24	26.0271
GLDfff	2.40909	±0.26	18.8487
Random	3.31818	±0.34	22.4312

Table 3.6: Derandomized and Randomized Samplers on the " Stock-Management", "Many-Bots", "Arm" and "Away" problems, with baseline dimension. One out of GLD and GLDfff samplers significantly dominates the others on "Stock-Management", "Many-Bots" and "Away"problems; In the case of Stock Management and Away, GLD is the best sampler whilst GLDff is the worst one; otherwise (Many-bots), GLD is the worst one while GLDff is the best one. On the Arm problem, randomized samplers are far better than derandomized ones.

figure 3.1 for small number of points, and [LL03, LBL04]). The GLD sampler dominates other blind samplers (low-dispersion approaches LD, GLDfff, RandomForAll, QR) on most problems, with stable performances overall; it is most appropriate when frontier is relevant (e.g. stock management problems) as it is biased toward the frontier of the search space. When the frontier of the search space is known to be irrelevant, GLDfff must be preferred to GLD[25]. Moreover, it is worth noting that GLD can be extended to accommodate any prior knowledge, e.g. to tailor the selection criterion (eq. 3.25; see Fig. 3.1). Further research will consider criteria suited to particular types of benchmark problems.

A second contribution focuses on non-blind active regression, specifically in high dimensions ($d \geq 8$). A specific *EA-sampler* has been devised, enforcing the following properties: i) *EA-sampler* achieves a tunable tradeoff between random sampling and greedy

[25]When the frontier is irrelevant, GLD is penalized as it sets 2^d points out of the first $2^d + 1$ points in the corners of $D = [0,1]^d$.

sampling depending on parameter α; ii) it works in high dimensions, whereas most active samplers do not scale up; iii) while it has been experimented with a specific fitness function (cost minimization), the fitness function can be tailored to accommodate prior knowledge (e.g. about penalties and constraint-based optimization).

Empirically, *EA-sampler* performances are unstable, sometimes excellent (Many-bots) and sometimes among the worst ones. Although such a lack of generality is disappointing, it is conjectured that non-blind methods require more parameter-tuning, than done in this section.

Overall, widely different performances are obtained depending on the sampler and on the problem. While we are still far from being able to recommend a given sampler for a given problem, the GLD blind sampler and the *EA-sampler* in high dimensions appear to be worth trying when tackling a new problem.

More generally, these results support the fact that active learning, and specifically, the use of sophisticated blind or non-blind samplers, is a key bottleneck for reinforcement learning and SDP, and deserves more extensive investigations.

Indeed, this study is meant to be complementary to the design of specific SDP samplers, e.g. focusing the search using prior knowledge-based policy [BBS93] or relying on dimensionality reduction techniques.

3.6 Summary

The three core tasks in Stochastic Dynamic Programming, namely optimization, learning and sampling, have been studied in a theoretical and empirical perspective.

The first contribution of the chapter is an extensive empirical study, which can be replicated and extended within the *OpenDP* ToolBox, with two goals in mind: enforcing fair comparisons (same benchmark problems, same assessment criteria, same time budget for hyperparameter tuning); measuring the "out-of-shelf" performance, with no use of prior knowledge or pre-processing (e.g. dimensionality reduction, problem decomposition) techniques. The dimension of the benchmark problems is small to moderate ($d \leq 12$).

Partial conclusions, detailed in the sections respectively devoted to Optimization,

Many Obst. Avoid. dim.x4 = 8		
Sampler	Average score	Std
EAS-0.2	80.90	±2.34
EAS-0.3	81.93	±2.25
EAS-0.4	87.95	±2.19
EAS-0.5	90.79	±1.97
EAS-0.6	89.77	±2.07
EAS-0.7	93.75	±1.68
EAS-0.75	91.02	±2.00
EAS-0.8	88.86	±2.18
EAS-0.85	87.04	±2.42
EAS-0.9	87.15	±2.15
EAS-0.95	80.22	±2.27
GLD	65	±1.50e-16

Arm dim.x4 = 12		
Sampler	Average score	Std
EAS-0.2	10.34	±0.21
EAS-0.3	10.17	±0.20
EAS-0.4	10.46	±0.23
EAS-0.5	10.62	±0.21
EAS-0.6	10.49	±0.20
EAS-0.7	10.54	±0.23
EAS-0.75	10.53	±0.22
EAS-0.8	10.37	±0.20
EAS-0.85	10.17	±0.20
EAS-0.9	10.44	±0.20
EAS-0.95	10.38	±0.21
GLD	12.52	±0.25

Away dim.x4 = 8		
Sampler	Average score	Std
EAS-0.2	0.38	±0.10
EAS-0.3	0.60	±0.15
EAS-0.4	0.42	±0.12
EAS-0.5	0.54	±0.15
EAS-0.6	0.47	±0.13
EAS-0.7	0.61	±0.19
EAS-0.75	0.57	±0.14
EAS-0.8	0.46	±0.11
EAS-0.85	0.55	±0.15
EAS-0.9	0.59	±0.17
EAS-0.95	0.69	±0.17
GLD	0.64	±0.17

Fast Obst. Avoid. dim.x4 = 8		
Sampler	Average score	Std
EAS-0.2	620.45	±17.53
EAS-0.3	726.13	±28.41
EAS-0.4	742.04	±28.75
EAS-0.5	822.72	±29.55
EAS-0.6	795.45	±29.87
EAS-0.7	812.5	±29.53
EAS-0.75	809.09	±28.86
EAS-0.8	815.90	±29.07
EAS-0.85	689.77	±25.05
EAS-0.9	729.54	±28.82
EAS-0.95	596.59	±11.87
GLD	550	±1.50e-16

Many bots dim.x4 = 8		
Sampler	Average score	Std
EAS-0.2	2295.23	±16.08
EAS-0.3	2311.59	±13.46
EAS-0.4	2179.09	±15.62
EAS-0.5	2175.45	±16.11
EAS-0.6	2156.59	±15.66
EAS-0.7	2137.5	±13.41
EAS-0.75	2167.95	±14.26
EAS-0.8	2195	±16.62
EAS-0.85	2184.09	±13.91
EAS-0.9	2170.23	±14.22
EAS-0.95	2245.91	±12.39
best-blind = GLD	2326.59	±16.17

Table 3.7: Non-Blind and Blind Samplers in high dimensions. *EA-sampler* good performances are observed for problems involving large bad regions (e.g. for the Many-bots problem); such regions can be skipped by non-blind samplers and must be visited by blind samplers. In some other problems, *EA-sampler* performances match those of random blind samplers (ROFCS) on the "Arm" and "Away" problems.

Learning and Sampling, suggest that some algorithms are more generally effective than others, although no algorithm was found to always dominate all others. The key factor of efficiency is the robustness of the algorithm, e.g. its ability to avoid dramatic failures over the few thousand problems handled along a RL run. Along these lines, *Evolutionary algorithms* for optimization, SVM for learning, and low dispersion sequences (GLD) for sampling, are the overall best algorithms in the range of the experiments. The main limitation of SVM is its computational cost with quadratic complexity in the number of examples. A limitation of GLD is its bias toward the domain frontiers (GLDfff should be preferred to GLD if the frontier domain is known to be irrelevant). *EA-sampler* might be considered as an alternative to GLD in higher dimensions ($d > 7$).

The second contribution of the chapter concerns the theoretical analysis of the sampling algorithms, more specifically their robustness (universal consistency) and their convergence rate. It has been shown that the UC property requires some randomness; but a limited amount of randomness, injected in derandomized sequences, is enough to warrant UC and good convergence rate for smooth target functions.

Chapter 4

High dimension discrete case: Computer Go

This chapter focuses on control in large discrete domains, exemplified by the Computer-Go domain. While the main challenges for continuous control are related to optimization, learning and sampling (SDP, Chapter 3), the challenge of discrete control is to manage an exploration vs exploitation dilemma, where i) not all actions can be tried; ii) not all states can be given a value.

The general problem of game playing, formalized as a discrete control problem, is briefly described in section 4.2, relying on two main functionalities: i) exploring a tree-structured search space; ii) learning the value function through repeated games (section 4.3).

In this perspective, our contribution, as demonstrated by the MoGo program[1], is twofold. The first contribution is to replace the global model of the value function adapted after every game, with an on-line stochastic value approximation process, adapted after every move (section 4.4). Surprisingly, a good quality strategy can be found with a relatively noisy value process − further, improving the policy strength in the prediction process process does not necessarily result in strategy improvements (section 4.4.2). Mathematical insights into this counter-intuitive result are presented in section 4.4.3.

[1] MoGo was the best computer-Go program at the time of writing. See Appendix C for the MoGo bibliography.

The second contribution presented in this chapter is related to the exploration of tree-structured search space, and inspired from the multi-armed bandit framework [ACBF02], specifically the UCT algorithm [KS06] (section 4.5). The basic idea is to take advantage of the domain structure, to generalize the value function to non visited state/action pairs. The Rapid Action Value Estimation (RAVE) algorithm is inspired from the "All Moves as first" heuristic in computer Go and adapted to UCT (section 4.5.6). The relevance of RAVE is significantly demonstrated as it increases the performance of the MoGo program, specifically in higher dimensions.

4.1 Reinforcement Learning and Games

Games traditionally are a source of applications for discrete optimal control and Reinforcement Learning. Some RL algorithms (e.g. $TD(\lambda)$ and variants, Chapter 1) perform at a master level the game of Checkers [SHJ01a], Othello [Bur99], and Chess [BTW98]. Two games remain unmastered: Go and Poker (see below).

Many RL-based approaches involve: i) a tree-structured exploration of the search space; ii) a value function, assessing the value of leaves and nodes in the tree. In many frameworks, each position is described as a valued vector, and the value function is characterized as a set of weights on the vector coordinates. After each game, the weights are updated; the policy proceeds by optimizing the value function. The knowledge learned (the set of weights) applies across the on-policy distribution of states.

The game of Go, an ancient Asian game enjoying a great popularity all over the world[2] induces two challenges, respectively regarding the tree-structured search, and the value function. On the one hand, while the rules of Go are simple (Fig. 4.1), the state space is large (the goboard ranges from 9×9 to 19×19) and the number of moves (branching factor of the Go tree search) is circa 300. In contrast, the rules of Chess are more complex, but the chess board is 8×8 and the branching factor is circa 40. On the other hand, no really efficient framework has been developed to model and learn the value function (see section 4.2). As a consequence, the alpha-beta search heuristics, meant to efficiently pruning the

[2]The interested reader is referred to `http://www.gobase.org` for a comprehensive introduction.

Game	Year of first program *Name of the program* (Author)	Status against human	Date of domination and/or *Name of the program*
Chess	1962 *Kotok-McCarthy*	$C > H$	1997 *Deeper Blue*
Backgammon	1979 *SKG 9.8* (Berliner)	$C > H$	1998 *TD-Gammon*
Checkers	1952 (Strachey, Samuel)	$C >> H$ (solved)	1994 *Chinook*
Otello	1982 *IAGO* (Rosenbloom)	$C >> H$	1997 *LOGISTELLO*
Scrabble	1977 (Shapiro and Smith)	$C > H$	1998 *MAVEN*
Bridge	1963 (Berlekamp)	$C \approx H$	*GIB*
Poker	1977 (Findler)	$C << H$	*POKI*
Go	1970 (Zobrist)	$C << H$	*MoGo*

Table 4.1: Status of some popular games in between the best human players and the best programs. First column is the name of the games. Second column is the year of the first "interesting" program playing this game, with sometimes the name of this program and the name of the author. Third column gives this current status between the best program (C) and the world champion (H). If the program is better than the best human, then the last column gives the year and the name of the program which beat the world champion the first time. If not, the last column gives the name of the current best program.

branches that are dominated in a minimax sense[3] does not perform well; this also contrasts with Chess, e,g, the DeepBlue algorithm heavily relies on alpha-beta search [New96]).

Both reasons explain why Go is now considered one of the most difficult challenges for Artificial Intelligence, replacing Chess in this role [BC01]. Table 4.1 depicts the status of programs vs Master human players in quite a few games. Presently, the best Computer-Go programs are at the level of average amateurs;

Both challenges of tree-exploration and value estimation will be tackled in MoGo, combining two advances respectively based on Monte-Carlo Go evaluation [Bru93a], and on tree-structured multi-armed bandits [KS06].

For the sake of completeness, these two advances will be presented respectively in sections 4.4 and 4.5.

[3]The minimax, or min-max, value of a position p, is the value of the final position of the game reached from p by two players playing "optimally", i.e. assuming that in each position one player will play the worst case for his opponent.

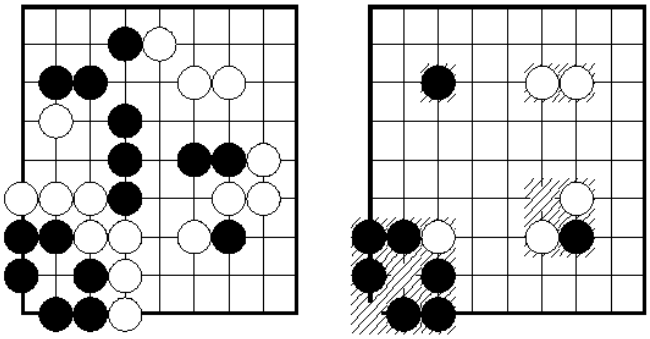

Figure 4.1: (Left) An example position from a game of 9×9 Go. Black and White take turns to place down stones. Stones can never move, but may be captured if completely surrounded. The player to surround most territory wins the game. (Right) Shapes are an important part of Go strategy. The figure shows (clockwise from top-left) 1×1, 2×1, 2×2 and 3×3 local shape features which occur in the example position.

4.2 A short computer-Go overview

We present in this section some classical methods used to build a program playing Go. The interested reader can find further details in the survey by Bouzy and Cazenave [BC01], or the survey by Müller [Mül02]. We present in the subsequent section, previous works more closely related to our approach, as Monte-Carlo evaluation method and Monte-Carlo tree search techniques.

While the first Go program was written by D. Lefkovitz [Lef60], the first Go program to beat a human player (an absolute beginner) was the program of Zobrist [Zob69, Zob70]. The work of Zobrist is also well know for his hashing function [Zob60] from a position to a number.

The first Go programs were only based on an influence function. Each stone on the board radiates influence to the surrounding intersections and so on, the radiation decreasing with the distance. These functions are still used in most Go programs as GoIntellect, Many Faces of Go or GnuGo.

The game of Go also has important properties which makes possible the cut of one position into (smaller) sub-problems. This leaded to the next generation of Go programs, e.g. Bruce Wilcox's one [Wil78] who divided board into zones.

The next age of computer go saw the use of patterns to recognize situations and suggest moves (e.g Goliat [Boo90]). Now many strong programs, which are called ”classical programs” use these techniques. On top of these, they use a tree search for tactical search, often only local but very fast, and for some also use slower global search.

More on more recent techniques as Monte-Carlo Go can be found in next section.

Life and Death problems

The life and death problems (Tsume-Go) are positions where the player has to state the status of strings. The status of a string can be alive (can't be captured) or dead (can be captured). These problems play a major role in the game of Go. Some works addressed specifically this, without trying to build a complete Go player. The most successful was Thomas Wolf's Gotools [Wol94, Wol00]. The achieved level is very strong. Gotools can solve 5-dan amateur problems, one of the top amateur level, close to professional level!

This level is very far from the level of computer players playing real games. That shows that even if life and death problems are important part of the game, it is not sufficient to achieve strong playing level, especially because Gotools is limited on enclosed position with few empty intersections.

Mathematical morphology

Looking at a Go board and a Go position, using image processing techniques sound appealing. Mathematical morphology [Ser82] has been applied successfully to the game of Go [Bou95b, Bou95a, Gnu99].

Let us give a very short introduction to mathematical morphology. A black/white image is defined e.g. by the set of black points. We call ”elements” these black points.

Let A a set of elements. Mathematical morphology is based on some basic operators:

- Dilation $D(A)$ is composed of A plus the neighboring elements of A;
- Erosion $E(A)$ is composed of A minus the neighboring elements of the complement of A;
- External boundary $ExtBound(A) = D(A) - A$;

- Internal boundary $IntBound(A) = A - E(A)$;
- Closing $Closing(A) = E(D(A))$;
- Opening $Opening(A) = D(E(A))$.

Opening and closing operators are the most useful operators. This can be applied to Go by assigning values to each intersection, starting from stones with positive value for black and negative for white, and applying these operators [Bou95b].

Combinatorial game theory

Classical game theory [JVN44] has not been as successful in Go as it has been in Chess, Checkers or Othello. Combinatorial game theory [Con76] tries to cut the global problem in simpler sub-problems.

[EB94] achieved great success in the late endgame positions (Yose). Using an mathematical abstractions of positions, their program ended games as the level of high ranked professional, sometimes even better. Everyone before thought that professional was playing the end game optimally.

4.3 Scope of the problem and Previous Related Works

4.3.1 Scope of the problem

Most game engines are made of a tree-structured search operator, and an evaluation function.

The evaluation function investigated in the following is a stochastic evaluation procedure (section 4.4); in each run the board to be evaluated undergoes a sequence of random moves, until arriving at a final position, and reporting the associated score (e.g. whether it is a winning position). The score averaged on some independent runs defines the value associated to the board, together with its confidence interval.

The above value procedure is used to guide the exploration of the tree-structured search space. Ideally, the exploration would follow a minimax tree search, pruning the dominated

branches. However, the available value function does hardly enable efficient pruning; as the number of runs is low for the sake of computational efficiency, the confidence intervals are large. Another framework for discrete decision based on uncertain information is therefore considered, namely the multi-armed bandit framework (section 4.3.4).

The global picture of the algorithm is displayed in Fig. 4.4.

Remark 1 *Although the search operator is commonly referred to as tree-structured, the search space is more an oriented graph (i.e. two different move sequences can lead to a same position). Following an early discussion about the Graph History Interaction Problem (GHI) [KM04], the distinction will be discarded in the remainder.*

4.3.2 Monte-Carlo Go

Monte-Carlo Go, designed by Bruegmann [Bru93a], increasingly attracts attention since its inception in 1993, in the Gobble program. It has been first turned into an evaluation procedure by Bouzy and Helmstetter [BH03], and it has been found surprisingly efficient, especially on 9×9 board. The best known Monte-Carlo Go based program, CrazyStone designed by Rémi Coulom [Cou06], won over many other Go programs endowed with domain knowledge[4], although CrazyStone itself has very little knowledge (e.g. it ignores the specific patterns used by e.g. GnuGo to evaluate a position).

A illustration and explanation of how Monte-Carlo evaluation function works in Go is given in Figure 4.2.

4.3.3 Monte-Carlo Tree Search

A Monte-Carlo Tree Search algorithm [CSB$^+$06, Cou06, KS06], is a best-first search algorithm, using Monte-Carlo as the evaluation function. The goal is to learn a tabular evaluation function specialized in one state (or position), function represented by a tree of subsequent states. The algorithm consists in a four-step loop, detailed below (see figure 4.3 from [CWB$^+$07] for an illustration). Notably, this is an any-time algorithm; the number of loops depends on the allotted resources.

[4]CrazyStone won the gold medal for the 9×9 Go game during the 11th Computer Olympiad at Turin 2006, beating several strong programs including GnuGo, Aya and GoIntellect.

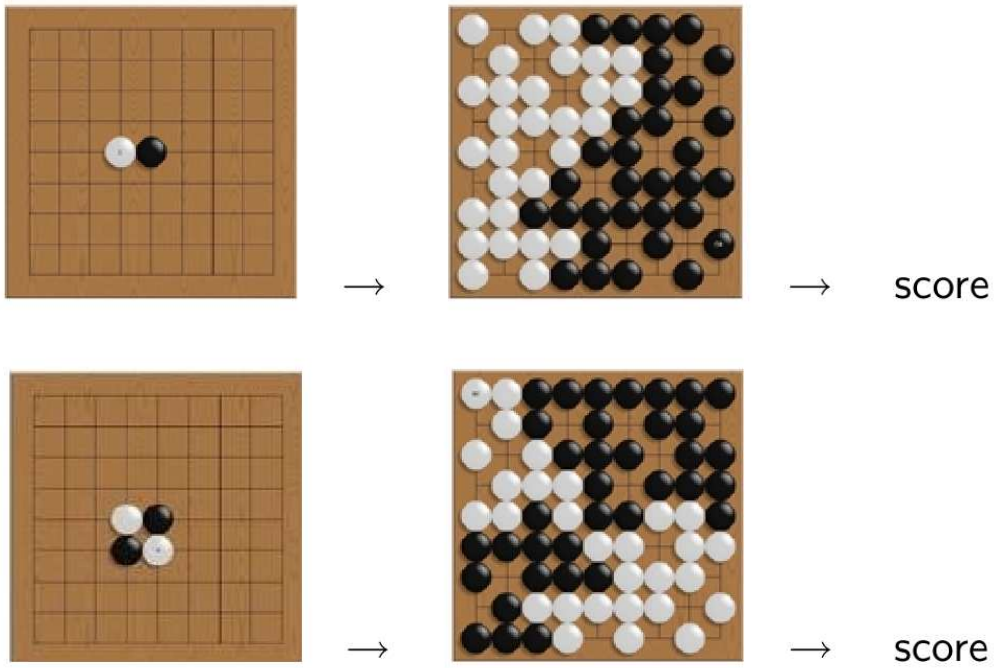

Figure 4.2: Illustration of Monte-Carlo evaluation function in Go. Given the position to be evaluated (left picture), and one simulation policy π (e.g. π playing uniformly randomly among legal moves except eyes), each run makes π play against itself until the end of the game (right picture). Every end position can be associated a score, computed exactly from the rules of the game; the score can also be a boolean (wins or loses). Overall, this procedure defines the value associated to the initial position as a random variable, e.g. a Bernoulli variable. One possibility is to run many times the Monte-Carlo evaluation to narrow the estimation after the Central Limit Theorem, and use (almost) deterministic decision policies. Another possibility is to handle the decision problem based on the available evidence and its confidence, along the multi-armed bandit framework (see section 4.5).

1. Start from the root and descend the tree until reaching a state which is not yet in the tree (*Selection* in Fig 4.3).

2. Depending on the expansion procedure, add the state as a new tree leaf (*Expansion* in Fig 4.3).

3. Run the simulation policy (e.g. play a random game) until arriving at some final position (*Simulation* in Fig 4.3) and compute the associated score.

4. According to this score value, update the value of all states on the tree path between the root node and the new leaf (*Backpropagation* in Fig 4.3)

4.3.4 Bandit Problem

As already mentioned, one specificity of the value procedure is to provide estimates of random variables. The decision problem is thus viewed as similar to the multi-armed bandit framework, defined as follows.

The core *exploration-exploitation* dilemma has been considered in many areas besides Reinforcement Learning, such as Game Theory. Specifically, this dilemma has been addressed in the so-called multi-armed bandit [ACBFS95], inspired from the traditional slot machines and defined as follows.

A multi-armed bandit involves n arms, where the i-th arm is characterized by its reward probability p_i. In each time step t, the gambler or the algorithm selects some arm j; with probability p_j it gets reward $r_t = 1$, otherwise $r_t = 0$. The loss after N time steps, or regret, is defined as $Np^* - E(\sum_{t=1}^{N} r_t)$, where p^* is the maximal reward probability among $p_1, \ldots, p_n$.

The objective of the gambler is to maximize the collected rewards or minimize the regret through iterative plays[5]. It is classically assumed that the gambler has no initial knowledge about the arms, but through repeated trials, he can focus on the most rewarding arms.

Two indicators are maintained for each i-th arm: the number of times it has been played up to time t, noted $n_{i,t}$ and the average corresponding reward noted $\hat{p}_{i,t}$. Subscript t is omitted when clear from the context.

The questions that arise in bandit problems are related to the problem of balancing reward maximization based on the knowledge already acquired and attempting new actions to further increase knowledge, which is known as the exploitation-exploration dilemma in reinforcement learning. Precisely, exploitation in bandit problems refers to select the

[5]We will use "play an arm" when referring to general multi-armed problems, and "play a move" when referring to Go. In Go application, the "play" will not refer to a complete game but only one move.

current best arm according to the collected knowledge, while exploration refers to select the sub-optimal arms in order to gain more knowledge about them.

The independence of the different arms, as well as the independence of rewards for a given arm, are traditionally assumed.

In Auer et Al. [ACBF02], a simple algorithm UCB1 is given, which ensures the optimal machine is played exponentially more often than any other machine uniformly when the rewards are in $[0,1]$.

Algorithm 1 *Deterministic policy: UCB1*

- *Initialization: Play each machine once.*
- *Loop: At time step t, play machine j that maximizes $\hat{p}_{j,t} + \sqrt{\frac{2\log n_t}{n_{j,t}}}$, where $n_t = \sum_j n_{j,t}$ is the overall number of plays done so far.*

Another formula with better experimental results is suggested in [ACBF02]. Let

$$V_{j,t} = \hat{p}_{j,t} - \hat{p}_{j,t}^2 + \sqrt{\frac{2\log n_t}{n_{j,t}}}$$

be an estimated upper bound on the variance[6] of machine j, then we have a new value to maximize:

$$\hat{p}_{j,t} + \sqrt{\frac{\log n_t}{n_{j,t}} \min\{1/4, V_{j,t}\}}\,. \tag{4.1}$$

According to Auer et al., the policy maximizing (4.1) named UCB1-TUNED, considering also the variance of the empirical value of each arms, performs substantially better than UCB1 in all experiments.

4.3.5 UCT

The UCT algorithm [KS06] is another value-based reinforcement learning algorithm. However, unlike other such algorithms it focuses its learning exclusively on the start state and the tree of subsequent states.

[6] The considered rewards are here 0 or 1 so $\hat{p}_{j,t}$ is also the average sum of squared rewards

The action value function $Q_{UCT}(s,a)$ is approximated by a partial tabular representation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A}$, containing a subset of all (state, action) pairs. This can be thought of as a search tree of visited states, with the start state at the root. A distinct value is estimated for each state and action in the tree by Monte-Carlo simulation.

The policy used by UCT is designed to balance exploration with exploitation, based on the multi-armed bandit algorithm UCB [ACBF02].

If all actions from the current state s are represented in the tree, $\forall a \in \mathcal{A}(s), (s,a) \in \mathcal{T}$, then UCT selects the action that maximises an upper confidence bound on the action value,

$$\begin{aligned} Q^{\oplus}_{UCT}(s,a) &= Q_{UCT}(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \\ \pi_{UCT}(s) &= \arg\max_a Q^{\oplus}_{UCT}(s,a) \end{aligned}$$

where $n(s,a)$ counts the number of times that action a has been selected from state s, and $n(s)$ counts the total number of visits to a state, $n(s) = \sum_a n(s,a)$.

If no action from the current state s is represented in the tree, $\exists a \in \mathcal{A}(s), (s,a) \notin \mathcal{T}$, then the uniform random policy π_{random} is used to select an action from all unrepresented actions, $\tilde{\mathcal{A}}(s) = \{a | (s,a) \notin \mathcal{T}\}$.

At the end of episode $s_1, a_1, s_2, a_2, ..., s_T$, each state action pair in the search tree, $(s_t, a_t) \in \mathcal{T}$, is updated using the score from that episode,

$$n(s_t,a_t) \leftarrow n(s_t,a_t) + 1 \tag{4.2}$$

$$\begin{aligned} Q_{UCT}(s_t,a_t) &\leftarrow Q_{UCT}(s_t,a_t) \\ &+ \frac{1}{n(s_t,a_t)}[R_t - Q_{UCT}(s_t,a_t)] \end{aligned} \tag{4.3}$$

New states and actions from that episode, $(s_t,a_t) \notin \mathcal{T}$, are then added to the tree, with initial value $Q(s_t,a_t) = R_t$ and $n(s_t,a_t) = 1$. In some cases, it is more memory efficient to only add the first visited state and action such that $(s_t,a_t) \notin \mathcal{T}$ [Cou06, GWMT06]. This procedure builds up a search tree containing n nodes after n episodes of experience.

The UCT policy can be thought of as a two-step policy. In the early stages of the episode, it visits the tree and selects actions on the basis of the knowledge associated to each node. After the last tree leaf is reached, no more knowledge is available, and the action selection proceeds randomly, until arriving at a final position and getting the associated value. Propagating this value up the tree, the value associated to each tree node becomes more precise, resulting in a better policy and focusing further the next Monte-Carlo simulations.

If a model is available, then UCT can be used as a sample-based search algorithm. Episodes are sampled from the model, starting from the actual state $\hat{s}$. A new representation $\mathcal{T}(\hat{s})$ is constructed at every actual time-step, using simulated experience. Typically, thousands of episodes can be simulated at each step, so that the value function contains a detailed search tree for the current state $\hat{s}$.

In a two-player game, the opponent can be modelled using the agent's own policy, and episodes simulated by self-play. UCT is used to maximise the upper confidence bound on the agent's value and to minimise the lower confidence bound on the opponent's. Under certain assumptions about non-stationarity, UCT converges on the minimax value [KS06]. However, unlike other minimax search algorithms such as alpha-beta search, UCT requires no prior domain knowledge to evaluate states or order moves. Furthermore, the UCT search tree is non-uniform and favours the most promising lines. These properties make UCT ideally suited to the game of Go, which has a large state space and branching factor, and for which no strong evaluation functions are known.

Pseudo-codes

For the sake of self-containedness, the UCT algorithm is described in Table 4.2[7]. Each simulation runs the playOneSequence routine (line 1 to line 8). The arm (or child node) selection is done after UCB1 (line 9 to line 21). Each arm must be selected at least once first (line 15). Line 16 computes the Upper Confidence Bound (UCB1) for each arm. After each sequence, all arms from the leaf to the root node are considered and their value is updated[8] iteratively using formula UCB1, described in function *updateValue* from line 22

[7]For clarity purpose, the code optimizations are not discussed here.

[8]Here we use the original formula in Algorithm 1.

to line 29. Here the code deals with the minimax case.

```
 1:  function playOneSequence(rootNode);
 2:      node[0] := rootNode; i = 0;
 3:      while(node[i] is not leaf) do
 4:          node[i+1] := descendByUCB1(node[i]);
 5:          i := i + 1;
 6:      end while ;
         node[i].value=endScore;
 7:      updateValue(node, -node[i].value);
 8:  end function;

 9:  function descendByUCB1(node)
10:      nb := 0;
11:      for i := 0 to node.childNode.size() - 1 do
12:          nb := nb + node.childNode[i].nb;
13:      end for;
14:      for i := 0 to node.childNode.size() - 1 do
15:          if node.childNode[i].nb = 0
                 v[i] := ∞;
16:          else v[i] := -node.childNode[i].value
             /node.childNode[i].nb
             +sqrt(2*log(nb)/(node.childNode[i].nb))
17:          end if;
18:      end for;
19:      index := argmax(v[j]);
20:      return node.childNode[index];
21:  end function;

22:  function updateValue(node,value)
23:      for i := node.size()-2 to 0 do
24:          node[i].value := node[i].value + value;
25:          node[i].nb := node[i].nb + 1;
26:          value := 1-value;
27:      end for;
28:  end function;
```

Table 4.2: Pseudocode of UCT for minimax tree. The ”endScore” is the value of the position for a leaf of the tree, which is here a final position.

4.3.6 Linear value function approximation

Instead of estimating the value of a position online (as in MC), one can learn offline a value function.

For a complex domain, such as Go, it is impractical to evaluate the value function of all states. Instead, various methods of function approximation have been tried [SDS94, Enz03, Sut96a]. [SSM07] has considered a simple approach that requires minimal prior domain knowledge, and which has proven successful in many other domains [BTW98, SHJ01a, Bur99].

We wish to estimate a simple reward function: $r = 1$ if the agent wins the game and $r = 0$ otherwise. The value function is approximated by a linear combination of binary features ϕ with weights θ,

$$Q_{RLGO}(s,a) = \sigma\left(\phi(s,a)^T\theta\right)$$

where the sigmoid squashing function σ maps the value function to an estimated probability of winning the game. After each time-step, weights are updated using the $TD(0)$ algorithm [Sut88]. Because the value function is a probability, the loss function is modified so as to minimise the cross entropy between the current value and the subsequent value,

$$\begin{aligned}\delta &= r_{t+1} + Q_{RLGO}(s_{t+1},a_{t+1}) - Q_{RLGO}(s_t,a_t)\\ \Delta\theta_i &= \frac{\alpha}{|\phi(s_t,a_t)|}\delta\phi_i\end{aligned}$$

where δ is the TD-error and α is a step-size parameter.

In the game of Go, the notion of *shape* has strategic importance. For this reason [SSM07] use binary features $\phi(s,a)$ that recognise local patterns of stones . Each *local shape feature* matches a specific configuration of stones and empty intersections within a particular rectangle on the board (Figure 4.1). Local shape features are created for all configurations, at all positions on the board, from 1×1 up to 3×3. Two sets of weights are used: in the first set, weights are shared between all local shape features that are rotationally or reflectionally symmetric. In the second set, weights are also shared between all

local shape features that are translations of the same pattern.

During training, two versions of the same agent play against each other, both using an ε-greedy policy. Each game is started from the empty board and played through to completion, so that the loss is minimised for the on-policy distribution of states. Thus, the value function approximation learns the relative contribution of each local shape feature to winning, across the full distribution of positions encountered during self-play.

4.4 Simulation Policy in Monte-Carlo

This section describes our first contribution to Monte-Carlo Go, based on a refined simulation policy hybridizing random policy and fast expert constraints. A second result, stemming from experimental studies, is that increasing the quality of the simulation policy does not necessarily result in a better quality of the control policy. A theoretical analysis for this unexpected finding, is proposed.

4.4.1 Sequence-like simulations

The baseline random simulation is based on the uniform selection of the moves among the legal ones, subject to a few rules preventing the program from filling its own eyes and favoring the moves capturing some stones.

A more educated random simulation is motivated to produce relevant sequences; basically, a set of patterns of interest inspired by Indigo [Bou05] (similar patterns can also be found in [RWB05]) is used to guide the selection of the new random move. While [BC06] and [Cou07] proposed methods to automatically learn interesting patterns, only $3x3$ manually defined patterns will be used in the following; more sophisticated patterns, e.g. as used in GnuGo, were not considered. A further constraint, referred to as locality, is introduced; it enforces the selection of a move contiguous (as long as it is possible) to the previous moves, resulting in much more realistic episodes.

The locality constraint was found to be very efficient empirically; it significantly improves on the only use of patterns to select the interesting moves. This surprising result will be further discussed in section 4.4.2.

Formally, a pattern is a 3×3 position, where the central intersection referred to as p is empty; the pattern is labelled as interesting or uninteresting (e.g. patterns detect the standard "cut move", "Hane move"). Any candidate position is matched against all available patterns (up to symmetry and rotations); its score is computed after the label of the patterns it matches.

Specifically, the educated random policy proceeds as follows. It first checks whether the last played move is an Atari[9]; if this is the case, and if the stones under Atari can be saved (in the sense that it can be saved by capturing stones or increasing liberties), it chooses one saving move randomly; otherwise it looks for interesting moves in the 8 positions around the last played move and plays one randomly if there is any. Finally, if still no move is found, it plays one move randomly on the Go board. The main frame is given here.

Figure 4.6 shows the first 30 moves of two games respectively played using the random and educated random policies. The latter one clearly results in a more realistic and meaningly position.

Formally, each pattern P is a 3×3 generalized position, represented as an element of $\{Full, Empty, Don't\ care\}^9$. A 3×3 position x is an element of $\{Full, Empty\}^9$; it strictly matches the pattern if $x_i < P_i$ for $i = 1 \dots 9$, with $Full < Don't\ care$ and $Empty < Don't\ care$; it matches the pattern if there exists a rotation or symmetry σ on x such that σx strictly matches P.

Candidate intersections define candidate positions (centered on the candidate intersections), which are tested against patterns if they are neither illegal nor self-Atari moves.

As already mentioned, we only used hand-coded patterns in MoGo, leaving the automatic learning of relevant patterns, possibly based on Bayesian learning [BC05], for further work. These patterns are shown in Fig. 4.7, 4.8, 4.9 and 4.10, where the position with a square is the candidate intersection. In addition to those patterns, the Atari moves are considered.

Remark 2 *We believe that it is not always better to have more 'good' patterns in the random modes, meanwhile what is more important is whether the random simulation can have some meaningful sequences often. This is developed in the following of this section.*

[9] *Atari* denotes the fact that a string has only one liberty left, so can be captured during the next move if nothing is done. This is an important notion of the game of Go.

The Table 4.3 shows clearly how patterns improve the overall performance. This simulation policy is called π_{MoGo} in the following.

Random mode	Win. Rate for B. Games	Win. rate for W. Games	Total Win. Rate
Uniform	46% (250)	36% (250)	$41.2\% \pm 2.2\%$
Sequence-like	77% (400)	82% (400)	$80\% \pm 1.4\%$

Table 4.3: Different modes with 70000 random simulations/move in 9x9.

4.4.2 Simulation player learned by $TD(\lambda)$

Previous part describes the introduction of prior knowledge to build an efficient simulation policy.

However, in many domains it is difficult to construct a good default policy. Even when expert knowledge is available, it may be difficult to interpret and encode. Furthermore, there is a practical requirement that policies must be fast to evaluate, otherwise the depth of the search tree will be limited by the time required to simulate episodes.

A linear combination of binary features provides one way to overcome these hurdles. [SSM07] learn a value function Q_{RLGO} offline, without prior domain knowledge (see Section 4.3.6). Furthermore, this representation is sufficiently fast for use in UCT search (see Table 4.10).

Monte-Carlo simulation works best when there is some random variation between episodes. We consider three different approaches for randomising the policy. First, we consider an ε-greedy policy,

$$\pi_\varepsilon(s,a) = \begin{cases} 1-\varepsilon+\frac{\varepsilon}{|A(s)|} & \text{if } a = \text{argmax}_{a'} Q_{RLGO}(s,a') \\ \frac{\varepsilon}{|A(s)|} & \text{otherwise} \end{cases}$$

Second, we consider a greedy policy over a noisy value function, corrupted by Gaussian noise $\eta \sim N(0,\sigma^2)$,

$$\pi_\sigma(s,a) = \begin{cases} 1 & \text{if } a = \text{argmax}_{a'} Q_{RLGO}(s,a') + \eta \\ 0 & \text{otherwise} \end{cases}$$

Third, we select moves using a softmax distribution with temperature parameter τ,

$$\pi_\tau(s,a) = \frac{e^{Q_{RLGO}(s,a)/\tau}}{\sum_{a'} e^{Q_{RLGO}(s,a')/\tau}}$$

We first compare the performance of each class of default policy π_{random}, π_{MoGo}, π_ε, π_σ, and π_τ. Figure 4.11 assesses the relative strength of each default policy (as a Go player), in a round-robin tournament of 6000 games between each pair of policies. With little or no randomisation, the policies based on Q_{RLGO} outperform both the random policy π_{random} and previously presented π_{MoGo} (see previous section) by a margin of over 90%. As the level of randomisation increases, the policies degenerate towards the random policy π_{random}.

Next, we compare the accuracy of each default policy π for Monte-Carlo simulation, on a test suite of 200 hand-labelled positions. 1000 episodes of self-play were played from each test position using each policy π. We measured the MSE between the average return (i.e. the Monte-Carlo estimate) and the hand-labelled value (see Figure 4.12). In general, a good policy for $UCT(\pi)$ must be able to evaluate accurately in Monte-Carlo simulation. In our experiments with *MoGo*, the MSE appears to have a close relationship with playing strength.

The MSE improves from uniform random simulations when a stronger and appropriately randomised default policy is used. If the default policy is too deterministic, then Monte-Carlo simulation fails to provide any benefits and the performance of π drops dramatically. If the default policy is too random, then it becomes equivalent to the random policy π_{random}.

Intuitively, one might expect that a stronger, appropriately randomised policy would outperform a weaker policy during Monte-Carlo simulation. However, the accuracy of π_ε, π_σ and π_τ never come close to the accuracy of the policy π_{MoGo}, despite the large improvement in playing strengths for these default policies. To verify that the default policies based on Q_{RLGO} are indeed stronger in our particular suite of test positions, we reran the round-robin tournament, starting from each of these positions in turn, and found that the relative strengths of the default policies remain similar. We also compared the performance of the complete UCT algorithm, using the best default policy based on Q_{RLGO} and the parameter minimising MSE (see Table 4.4). This experiment confirms that the MSE results apply in

Algorithm	Wins .v. GnuGo
$UCT(\pi_{random})$	8.88 $\pm$ 0.42 % (3420)
$UCT(\pi_{\sigma})$	9.38 $\pm$1.9% (224)
$UCT(\pi_{MoGo})$	48.62 $\pm$1.1% (2520)

Table 4.4: Winning rate of the UCT algorithm against GnuGo 3.7.10 (level 0), given 5000 simulations per move, using different default policies. The numbers after the $\pm$ correspond to the standard error. The total number of complete games is shown in parentheses. π_{σ} is used with $\sigma = 0.15$

actual play.

It is surprising that an objectively stronger default policy does not lead to better performance in UCT. Furthermore, because this result only requires Monte-Carlo simulation, it has implications for other sample-based search algorithms. It appears that the nature of the simulation policy may be as or more important than its objective performance. Each policy has its own bias, leading it to a particular distribution of episodes during Monte-Carlo simulation. If the distribution is skewed towards an objectively unlikely outcome, then the predictive accuracy of the search algorithm may be impaired.

This suprising result is investigated analytically, relating the strength of the simulation policy and the accuracy of the Monte-Carlo estimate in next section.

4.4.3 Mathematical insights: *strength* VS *accuracy*

This section investigates the relationship between the strength of a simulation policy used in Monte-Carlo and the accuracy of the estimation. We first prove that the policy quality reflects the estimation accuracy when the policy only has access to the current position, whereas it does not hold when some limited memory of the game is available to the policy.

These results are consistent with the fact that the locality constraint implemented in π_{MoGo} (only considering candidate intersections near to previous moves) which can be viewed as a limited memory about the game, significantly improves the overall results.

Notations

Assume a game with two players. Let $\mathcal{S}$ the finite set of states. Let $\mathcal{A}$ the finite set of moves. Let $p(s'|s,a)$ the transition probability function. In a deterministic game as the game of Go, for all $s,s' \in \mathcal{S}$ there exists at most one action a where $p(s'|s,a) > 0$ (and in this case $p(s'|s,a) = 1$).

We note $\Omega(\mathcal{A})$ the set of random variables on $\mathcal{A}$ (set of functions from the probability space Ω to $\mathcal{A}$). Let π a policy $\pi : \mathcal{S} \to \Omega(\mathcal{A})$. π can be stochastic, so $\pi(s)$ is a random variable. For π a policy, and $s \in \mathcal{S}$, let $Im(\pi(s)) = \{a \in \mathcal{A}, P(\pi(s) = a) > 0\}$. So $Im(\pi(s))$ is the set of action which can be chosen by π on the state s.

For a real value v and a real $\varepsilon > 0$, we note $v \pm \varepsilon = [v - \varepsilon, v + \varepsilon]$

Let $c : \mathcal{S} \to \{1,2\}$ the application giving the player to play in a given state. We assume that the players alternate in the game, i.e. $\forall (s,s') \in \mathcal{S}^2, \forall a \in \mathcal{A}, c(s) = c(s') \Longrightarrow p(s'|s,a) = 0$. We assume that there is no infinite games, which is equivalent to assume that it is impossible to reach twice a position in a game. Then, the directed graph $(\mathcal{S}, \mathcal{E})$ with $\mathcal{E} = \{(s,s') \in \mathcal{S}^2, \exists a \in \mathcal{A},\ p(s'|s,a) \neq 0\}$ is acyclic. We note $\mathcal{T}$ the set of terminal states, i.e. $\mathcal{T} = \{s \in \mathcal{S}, \forall s' \in \mathcal{S}, \forall a \in \mathcal{A}, p(s'|s,a) = 0\}$. Taking the topological order in this graph, we can number all the states, starting for the terminal states. We note s_i the i^{th} state in this topological order. Let $score : \mathcal{T} \times \{1,2\} \to \mathbb{R}$ the function giving the score of the game for terminal positions for each player. This score function can be arbitrary in games which give only a winner and a looser (e.g. 1 for a winning game, 0 for a loosing game).For $s \in \mathcal{T}$, the first player wins iff $score(s,1) > score(s,2)$.

Given two players π_1 and π_2, and a position $s \in \mathcal{S}$ we note $P_s(\pi_1 \to \pi_2)$ the probability of winning for π_1 against π_2 in the position s (π_1 plays first in s). If $\forall s, P_s(\pi_1 \to \pi_2) = P_s(\pi_1 \to \pi_1) = P_s(\pi_2 \to \pi_2)$, then we will say that π_1 and π_2 has the same *strength*.

The Monte-Carlo method takes a simulation policy π and approximate $P_s(\pi \to \pi)$ by $\hat{P}_s(\pi \to \pi)$ the empirical average of results given a finite number of roll out games from s using π as a player. Then the value of the position is $\hat{P}_s(\pi \to \pi)$. By the law of large numbers, $\hat{P}_s(\pi \to \pi) \longrightarrow P_s(\pi \to \pi)$ with the number of playouts. We also have that $\hat{P}_s(\pi \to \pi)$ is an unbiased estimator of $P_s(\pi \to \pi)$, i.e. $E\ \hat{P}_s(\pi \to \pi) = P_s(\pi \to \pi)$.

For a player Π, It is natural to consider that $P_s(\Pi \to \Pi)$ is the *value* of the position s for

Π as it is the probability of winning of Π against itself. Indeed, Π expect an opponent of its strength, then its probability of winning on a position is exactly $P_s(\Pi \to \Pi)$. The fact that "Π expect an opponent of its strength" is true for most of the search algorithms in games, where the max and min levels are modeled by the same algorithm and same evaluation function.

Theorems

Theorem 4.4.1 (Accurate simulation player without memory is strong) *Let* $\Pi : \mathcal{S} \to \Omega(\mathcal{A})$ *a reference player and* $\pi : \mathcal{S} \to \Omega(\mathcal{A})$ *a simulation player.*

If

$$\forall s \in \mathcal{S}, P_s(\Pi \to \Pi) = P_s(\pi \to \pi)$$

then

$$\forall s \in \mathcal{S}, P_s(\Pi \to \pi) = P_s(\pi \to \Pi) = P_s(\Pi \to \Pi)$$

.

Interpretation: We are here in the case where the simulation player π use only the position to choose its moves (possibly stochastic), because π depends only on the state.

Then the theorem says that if the estimated value (by Monte-Carlo) of each position ($P_s(\pi \to \pi)$ which is the limit of the Monte-Carlo estimator by the law of large numbers, and also the expectation of this estimator) is exactly the value of the position for Π, then the simulation policy π has the same strength as Π.

That also means that you can't make a simulation policy which is accurate for Π if you don't make it as strong as Π.

Corollary 1 (Simulation players with same accuracy are equally strong) *If you now consider* π *and* Π *as two simulation players, the previous theorem says that if their estimation of all positions are equal, then they have the same strength.*

Proof Let us show by induction on the topological order of positions that for each position s, $P_s(\Pi \to \pi) = P_s(\pi \to \Pi) = P_s(\Pi \to \Pi)$.

For $s \in \mathcal{T}$, the statement is obviously true, as the game is finished and the score does not depend on the player. Let $n > |\mathcal{T}|$ and assume the property true for all $s_i \in \mathcal{S}$ with $i < n$.

We have:

$$P_{s_n}(\pi \to \pi) = 1 - E_{\pi(s_n)} P_{\pi(s_n)}(\pi \to \pi) \tag{4.4}$$

$$P_{s_n}(\Pi \to \Pi) = 1 - E_{\Pi(s_n)} P_{\Pi(s_n)}(\Pi \to \Pi) \tag{4.5}$$

$$P_{s_n}(\pi \to \Pi) = 1 - E_{\pi(s_n)} P_{\pi(s_n)}(\Pi \to \pi) \tag{4.6}$$

$$P_{s_n}(\Pi \to \pi) = 1 - E_{\Pi(s_n)} P_{\Pi(s_n)}(\pi \to \Pi) \tag{4.7}$$

From the topological order of positions, we can apply the induction property to the $s \in Im(\pi(s_n))$ and the $s \in Im(\Pi(s_n))$. Then, from formulas 4.4 and 4.6, $P_{s_n}(\pi \to \pi) = P_{s_n}(\pi \to \Pi)$. From formulas 4.5 and 4.7, $P_{s_n}(\Pi \to \Pi) = P_{s_n}(\Pi \to \pi)$. As from hypothesis, we have $P_{s_n}(\pi \to \pi) = P_{s_n}(\Pi \to \Pi)$, the property is shown for s_n. The induction concludes the proof. □

Theorem 4.4.2 (Almost accurate simulation player without memory is almost strong) *Let $\Pi : \mathcal{S} \to \mathcal{A}$ a reference player and $\pi : \mathcal{S} \to \mathcal{A}$ a simulation player. Let $\varepsilon > 0$.*

If

$$\forall s \in \mathcal{S}, P_s(\Pi \to \Pi) = P_s(\pi \to \pi) \pm \varepsilon$$

then

$$\forall s_i \in \mathcal{S}, P_{s_i}(\Pi \to \pi) = P_{s_i}(\Pi \to \Pi) \pm i\varepsilon$$

,

$$P_{s_i}(\pi \to \Pi) = P_{s_i}(\Pi \to \Pi) \pm i\varepsilon$$

,

$$P_{s_i}(\Pi \to \pi) = P_{s_i}(\pi \to \pi) \pm i\varepsilon$$

and

$$P_{s_i}(\pi \to \Pi) = P_{s_i}(\pi \to \pi) \pm i\varepsilon$$

.

Interpretation: As in the previous theorem, if the simulation player, having access only

to the states, accurately estimates each position, then it is strong. However the error ε in the estimates can accumulate and this accumulation grows linearly in the distance to the end positions.

Remark 1: this is a worst case.

Remark 2: this result can be refined. The i in the $i\varepsilon$ bound, instead of begin the number of the position in the topological order, can be the maximum number of moves to reach an end position using the given policies. This can be much better, as for a starting position, i is here the number of total possible position in the game, though it could simply approximately be the number of moves of the game. However the proof would be very similar, with only technical complications.

Proof The proof follows closely the previous theorem proof. Let us show by induction on the topological order of positions that for each position s_i, $(\Pi \to \pi) = P_{s_i}(\Pi \to \Pi) \pm i\varepsilon$ and $P_{s_i}(\pi \to \Pi) = P_{s_i}(\Pi \to \Pi) \pm i\varepsilon$.

For $s \in \mathcal{T}$, the statement is obviously true, as the game is finished and the score does not depend on the player. Let $n > |\mathcal{T}|$ and assume the property true for all $s_i \in \mathcal{S}$ with $i < n$.

The formulas 4.4, 4.6, 4.5 and 4.7 still hold.

From the topological order of positions, we can apply the induction property to the $s \in Im(\pi(s_n))$ and the $s \in Im(\Pi(s_n))$. Then, from formulas 4.4 and 4.6, $P_{s_n}(\pi \to \pi) = P_{s_n}(\pi \to \Pi) \pm (n-1)\varepsilon$.

From formulas 4.5 and 4.7, $P_{s_n}(\Pi \to \Pi) = P_{s_n}(\Pi \to \pi) \pm (n-1)\varepsilon$.

As from hypothesis, we have $P_{s_n}(\pi \to \pi) = P_{s_n}(\Pi \to \Pi) \pm \varepsilon$,

$P_{s_i}(\Pi \to \pi) = P_{s_i}(\Pi \to \Pi) \pm i\varepsilon$, $P_{s_i}(\pi \to \Pi) = P_{s_i}(\Pi \to \Pi) \pm i\varepsilon$, $P_{s_i}(\Pi \to \pi) = P_{s_i}(\pi \to \pi) \pm i\varepsilon$ and $P_{s_i}(\pi \to \Pi) = P_{s_i}(\pi \to \pi) \pm i\varepsilon$.

the property is shown for s_n. The induction concludes the proof.

□

The previous results seems to argue that if you want to improve the evaluation function by Monte-Carlo, then you have to make the simulation policy stronger. That seems quite obvious. This is however in contradiction with the experimental results. We will now see that this result does not hold anymore if you give some (even very limited) memory to the simulation policy.

Memory allows the simulation player to be weak

Let us consider the following game. There are two players, who alternate, and one counter variable V which starts with value 0. There is a fixed number of turns $n \in \mathbb{N}$, with n even.At his turn, the player gives a value $v \in [-1,1]$ and the counter V becomes $V \leftarrow V + v$. After n moves (n is part of the definition of the game), the game stops, and if $V > 0$ then the first player wins, else ($V \leq 0$) the second player wins.

Formally, let $n \in \mathbb{N}$, then $\mathcal{S} = [0,n] \times \mathbb{R}$, and $\mathcal{A} = [-1,1]$. The probability transition is defined as: let $s = [i,v] \in \mathcal{S}$, $s' = [i',v'] \in \mathcal{S}$ and $a \in \mathcal{A}$, then $p(s'|s,a) = 1$ if and only if $i' \geq 1$, $i' = i-1$ and $v' = v+a$. Here $\mathcal{T} = \{(0,v)|v \in [-n,n]\}$.

Of course an optimal strategy for this game is obvious: the first player has to play always 1 and the second -1. As we assume n even, the game between two optimal players is a draw.

This game can be seen as an idealised version of number of games, where the counter variable V is the current score of the position, and at the end of the game, the player with the highest score wins the game. One can also model why the Monte-Carlo evaluation function works in the game of Go with this game, seeing V as the current expected score, and the simulation player making a random walk, adding or subtracting by his (good or bad) moves some points to the score.

To simplify the analysis, we take a deterministic player (with a stochastic player we will have the same result, but the technical discussion would be more complicated), let us say the optimal player. So Π is such as

$$\Pi((i,v)) = \begin{cases} -1 & \text{if } i \text{ is odd,} \\ 1 & \text{if } i \text{ is even.} \end{cases}$$

We saw in previous theorems that if we want to make a simulation policy π able to estimate each position accurately according to Π, π would have to be as good as Π, so here optimal. We will now construct a π which will have access to some very limited memory, much weaker than Π, but which will be a perfect estimator by Monte-Carlo.

The memory will be called $m \in \{0,1\}$, initialised to 0. Then π (depending on the state (i,v) and m) is defined as (with $\varepsilon > 0$):

$$\pi((i,v),m) = \begin{cases} -\varepsilon & \text{if } i \text{ is odd and } m = 1, \\ \varepsilon & \text{if } i \text{ is even and } m = 1, \\ -1 & \text{if } i \text{ is odd and } m = 0. \text{ In this case } m \leftarrow 1. \end{cases}$$

With $\varepsilon < 1$, then π is obviously much weaker than Π, but in each state, the evaluation by Monte-Carlo using π as the simulation policy is exactly the same as the value of the state (relative to Π).

Discussion on these results

These mathematical results are not totally satisfactory for several reasons. The first theorems, assess results between the strength of a policy and the accuracy of the estimate, but these results only hold when some equality or almost equality in all positions exist. It would be better to have inequality relationships between the strength and accuracy. For example, if Π_2 is stronger than Π_1, and π_1 estimates accurately the value for Π_1, then π_2 would have to be stronger than π_1 to estimate accurately Π_2.

Also, the example showing that we can build a weak simulation player which estimates accurately positions if it has access to some memory does not tell how to build a good simulation policy in real cases.

However these results are still interesting because they try to formalise the surprising experimental results. With this simple formalisation, we can hope finding ways to build very accurate simulation policy without having to make them strong. This a great hope, because building a strong simulation policy is as difficult as building a strong player.

4.5 Tree search and improvements in UCT algorithm

This section describes the use of the UCT algorithm for discrete control in the Go framework, discussing its pros and cons comparatively to alpha-beta search. The extensions of UCT, tailored to control, are thereafter presented and constitute the second contribution of this chapter. Incidentally, MoGo was the first Go program to involve the UCT algorithm

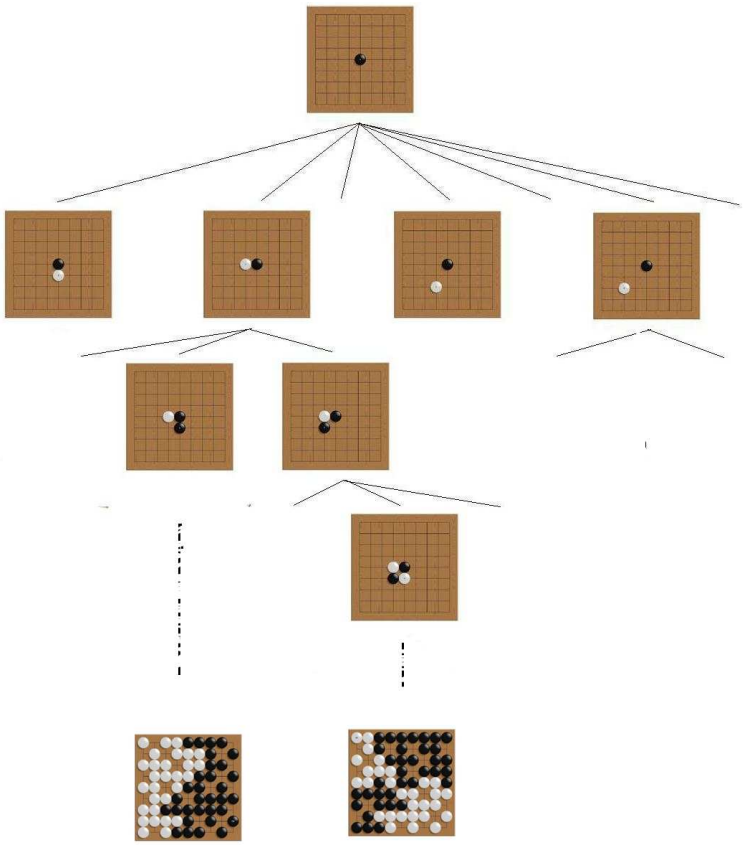

Figure 4.3: Illustration of the two parts of the algorithm, namely the tree search part and the Monte-Carlo part. The root represents the analysed position. Each node represents a position of the game, each branch leading from a position to another playing exactly one move. The tree search part is done using the UCT algorithm. The Monte-Carlo evaluation function consists in starting from the position and playing with a *simulation policy* until the end of the game. Then the game is scored exactly simply using the rules.

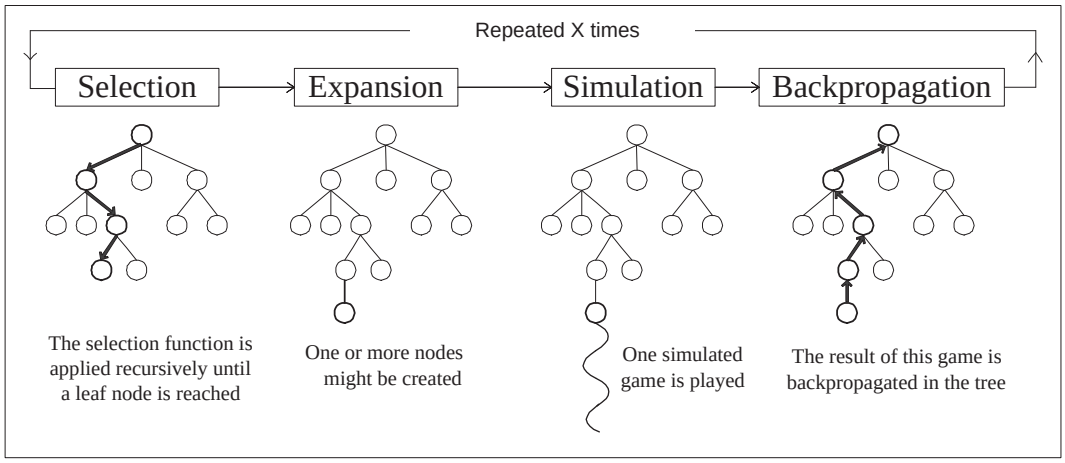

Figure 4.4: Illustration of a Monte-Carlo Tree search algorithm, figure from [CWB+07].

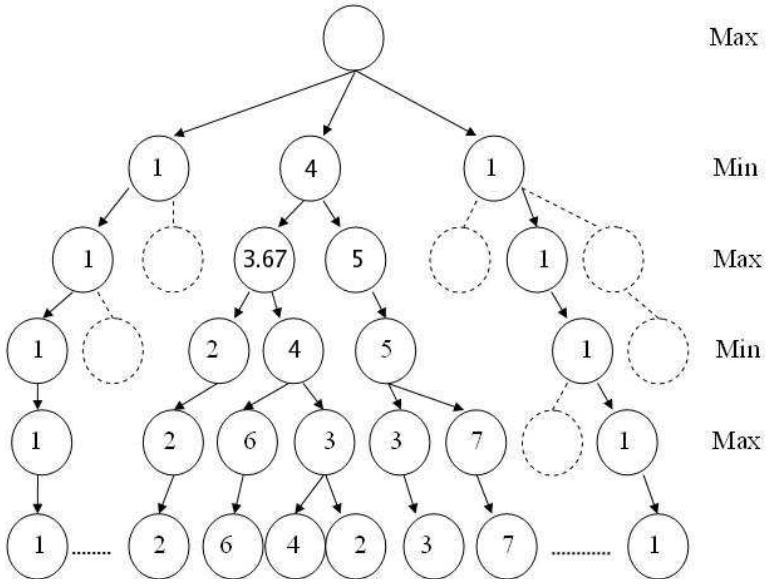

Figure 4.5: UCT search. The shape of the tree enlarges asymmetrically. Only updated values ($node[i].value$) are shown for each visited nodes.

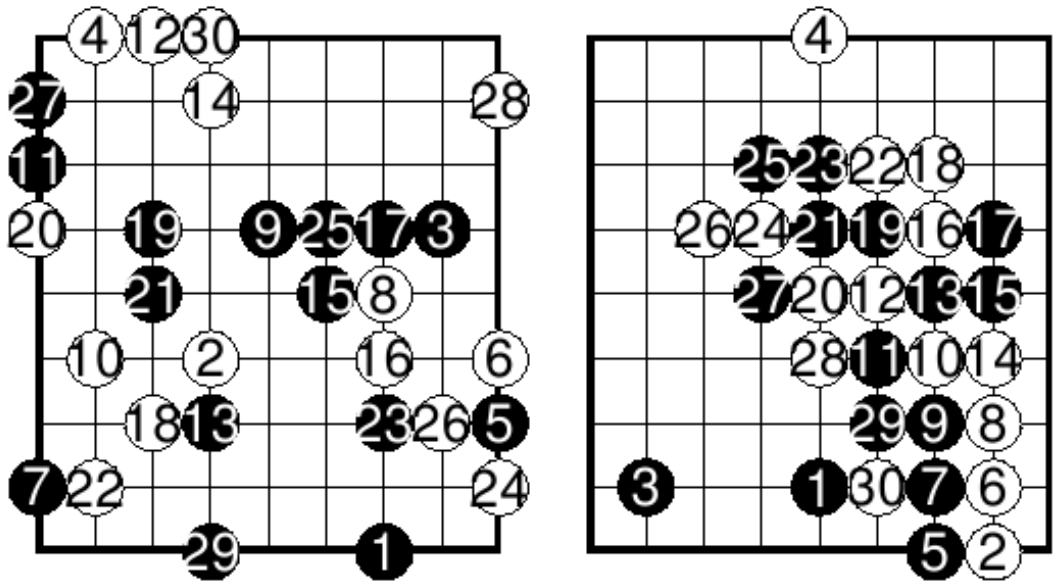

Figure 4.6: Comparing the random (Left) and educated random (Right) policies (first 30 moves). All 30 moves in the left position and the first 5 moves of the right position have been played randomly; moves 6 to 30 have been played using the educated random policy in the right position.

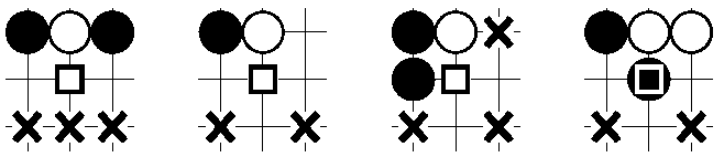

Figure 4.7: Patterns for Hane. True is returned if any pattern is matched. In the right one, a square on a black stone means true is returned if and only if the eight positions around are matched and it is black to play.

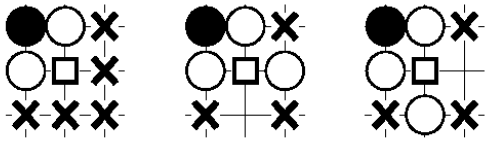

Figure 4.8: Patterns for Cut1. The Cut1 Move Pattern consists of three patterns. True is returned when the first pattern is matched and the next two are not matched.

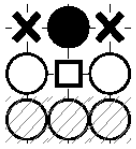

Figure 4.9: Pattern for Cut2. True is returned when the 6 upper positions are matched and the 3 bottom positions are not white.

Figure 4.10: Patterns for moves on the Go board side. True is returned if any pattern is matched. In the three right ones, a square on a black (resp. white) stone means true is returned if and only if the positions around are matched and it is black (resp. white) to play.

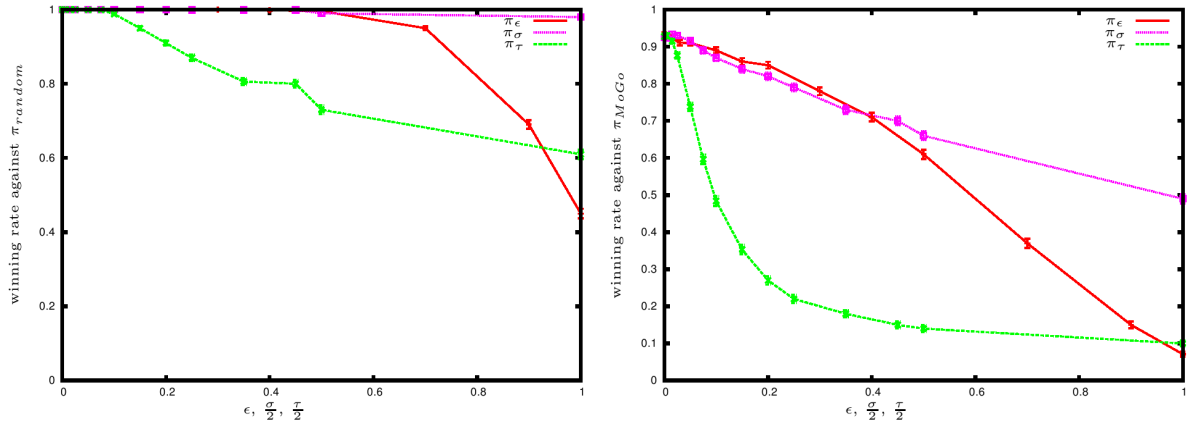

Figure 4.11: The relative strengths of each class of default policy, against the random policy π_{random} (left) and against the policy π_{MoGo} (right).

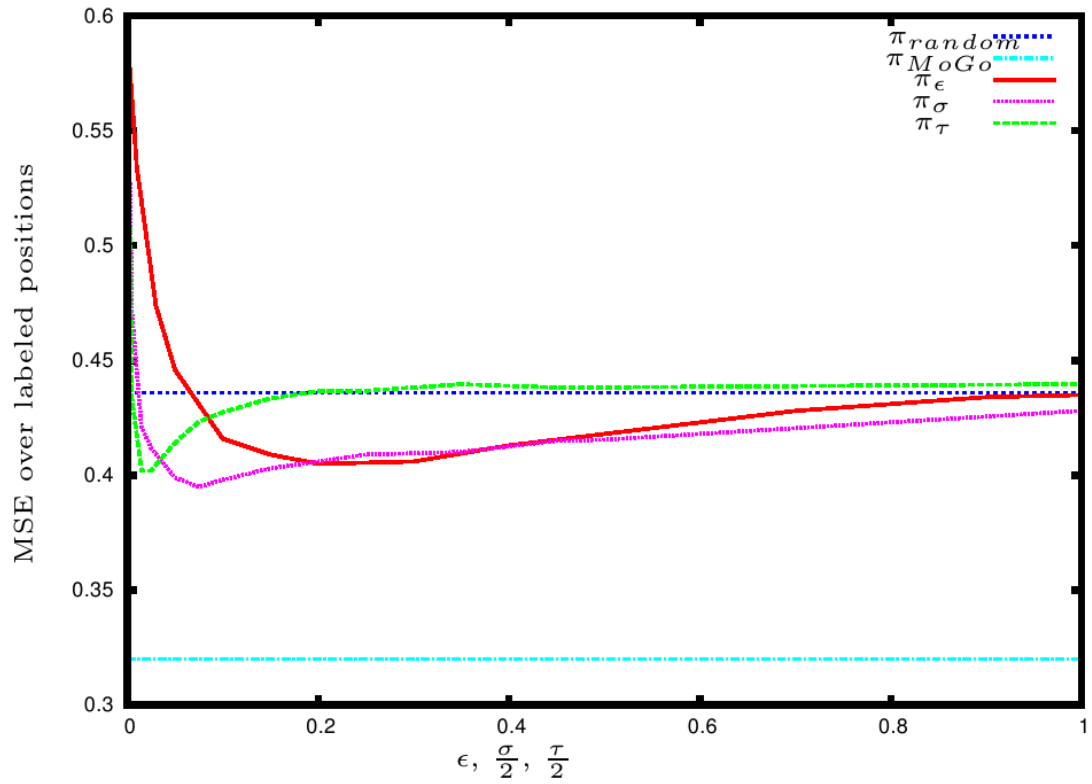

Figure 4.12: The MSE of each policy π when Monte Carlo simulation is used to evaluate a test suite of 200 hand-labelled positions.

whereas most of the strongest Go programs are now using UCT[10].

Let us first note that UCT is naturally closely related to Monte-Carlo; when UCT encounters a state that is not represented in its search tree, it proceeds randomly and thus calls a default random policy. A natural extension to the UCT algorithm is to use a *default policy* that incorporates general domain knowledge, such as policy π_{MoGo} defined in section 4.4.1. In the following, the original UCT algorithm is denoted $UCT(\pi_{random})$; UCT with default policy π is denoted $UCT(\pi)$.

4.5.1 UCT in the game of Go

As already mentioned (Figure 4.13), the complete algorithm is made of a tree search operator and a random simulation procedure. Each node of the tree represents a Go board situation, with child-nodes representing next situations after corresponding move.

The use of UCT for move selection relies on the equivalence between a Go position and a bandit problem, where each legal move corresponds to an arm, the associated reward of which is a random variable with unknown distribution. Let us consider in the following the case where every arm either is a winning one (reward 1) or a loosing one (reward 0). Cases of draw, almost nonexistent in Go, are ignored.

In the tree search part, a parsimonious version of UCT is used, based on a dynamic tree structure inspired from the 2006 version of CrazyStone [Cou06]. The tree is then created incrementally by adding one node after each simulation (see below). While this procedure is different from the one presented in [KS06], it is more efficient as less nodes are created during simulations. In other words, only nodes visited more than twice are saved, which economizes largely the memory and accelerates the simulations. The pseudocode is given in Table 4.5.

During each simulation, MoGo starts from the root of the tree that it saves in the memory. At each node, MoGo selects one move according to the UCB1 formula 4.1. MoGo then descends to the selected child node and selects a new move (still according to UCB1) until such a node has not yet been created in the tree. This part corresponds to the code

[10] Very recently [Cou07] combined offline learned knowledge into UCT. Combining these improvements with the improved UCT presented in this section is a perspective for further research.

```
1:  function playOneSequenceInMoGo(rootNode)
2:      node[0] := rootNode; i := 0;
3:      do
4:          node[i+1] := descendByUCB1(node[i]); i := i + 1;
5:      while node[i] is not first visited;
6:      createNode(node[i]);
7:      node[i].value := getValueByMC(node[i]);
8:      updateValue(node,-node[i].value);
9:  end function;
```

Table 4.5: Pseudocode of parsimonious UCT for MoGo

from line 1 to line 5. The tree search part ends by creating this new node (in fact one leaf) in the tree. This is finished by *createNode*. Then MoGo calls the random simulation part, the corresponding function *getValueByMC* at line 7, to give a score of the Go board at this leaf.

In the random simulation part, one random game is played from the corresponding Go board till the end, where score is calculated quickly and precisely according to the rules. The nodes visited during this random simulation are not saved. The random simulation done, the score received, MoGo updates the value at each node passed by the sequence of moves of this simulation[11].

Remark 3 *In the update of the score, we use the* 0/1 *score instead of the territory score, since the former is much more robust. Then the real minimax value of each node should be either* 0 *or* 1. *In practice, however, UCT approximates each node by a weighted average value in* $[0,1]$. *This value is usually considered as the probability of winning.*

4.5.2 Advantages of UCT compared to alpha-beta in this application

In the problems of minimax tree search, what we are looking for is often the optimal branch at the root node. It is sometimes acceptable if one branch with a score near to the optimal one is found, especially when the depth of the tree is very large and the branching factor is

[11] It is possible to arrive at one end game situation during the tree search part. In this case, one score could be calculated immediately and there is no need to create the node nor to call the random simulation part.

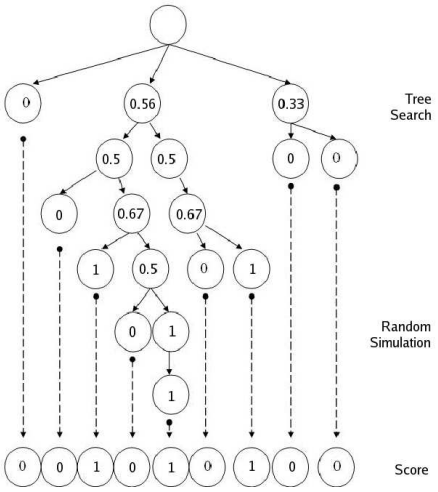

Figure 4.13: MoGo contains the tree search part using UCT and the random simulation part giving scores. The numbers on the bottom correspond to the final score of the game (win/loss). The numbers in the nodes are the updated values of the nodes $(node[i].value/node[i].nb)$
.

big, like in Go, as it is often too difficult to find the optimal branch within short time.

In this sense, UCT outperforms alpha-beta search. Indeed we can outlight three major advantages. First, it works in an anytime manner. We can stop at any moment the algorithm, and its performance can be somehow good. This is not the case of alpha-beta search. Figure 4.14 shows if we stop alpha-beta algorithm prematurely, some moves at first level has even not been explored. So the chosen move may be far from optimal. Of course iterative deepening can be used, and solve partially this problem. Still, the anytime property is stronger for UCT and it is easier to finely control time in UCT algorithm.

Second, UCT is robust as it automatically handles uncertainty in a smooth way. At each node, the computed value is the mean of the value for each child weighted by the frequency of visits. Then the value is a smoothed estimation of max, as the frequency of visits depends on the difference between the estimated values and the confidence of this estimates. Then, if one child-node has a much higher value than the others, and the estimate is good, this child-node will be explored much more often than the others, and then UCT selects most of the time the 'max' child node. However, if two child-nodes have a similar value, or a

low confidence, then the value will be closer to an average.

Third, the tree grows in an asymmetric manner. It explores more deeply the good moves in an adaptive manner. Figure 4.5 page 161 gives an example.

Figure 4.5 and Figure 4.14 compares the explored tree of two algorithms within limited time. However, the theoretical analysis of UCT is in progress [KSW06]. We just give some remarks on this aspect at the end of this section. It is obvious that the random variables involved in UCT are not identically distributed nor independent. This complicates the analysis of convergence.

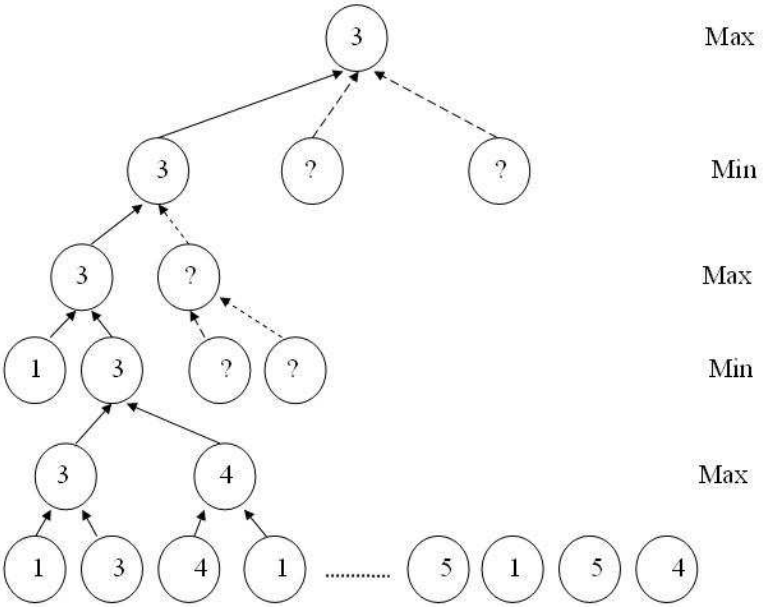

Figure 4.14: Alpha-beta search with limited time. The nodes with '?' are not explored yet. This happens often during the large-sized tree search where entire search is impossible. Iterative deepening solves partially this problem.

4.5.3 Exploration-exploitation influence

We parametrize the UCT implemented in our program a parameter, namely p. We add one coefficient p to formula UCB1-TUNED (4.1), which by default is 1. This leads to the

following formula: choose j that maximizes:

$$\bar{X}_j + p\sqrt{\frac{\log n}{T_j(n)}\min\{1/4, V_j(n_j)\}}$$

p decides the balance between exploration and exploitation. To be precise, the smaller the p is, the deeper the tree is explored. According to our experiment shown in Table 4.6, UCB1-TUNED is almost optimal in this sense.

p	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
0.05	2% ± 2% (50)	4% ± 2.5% (50)	3% ± 1.7%
0.55	30% ± 6.5% (50)	36% ± 6.5% (50)	33% ± 4.7%
0.80	33% ± 4.5% (100)	39% ± 5% (100)	36% ± 3.3%
1.0	40% ± 4% (150)	38% ± 4% (150)	39% ± 2.8%
1.1	39% ± 4% (150)	41% ± 4% (150)	40% ± 2.8%
1.2	40% ± 4% (150)	44% ± 4% (150)	42% ± 2.9%
1.5	30% ± 6.5% (50)	26% ± 6% (50)	28% ± 4.5%
3.0	36% ± 6.5% (50)	24% ± 6% (50)	30% ± 4.5%
6.0	22% ± 5.5% (50)	18% ± 5% (50)	20% ± 4%

Table 4.6: Coefficient p decides the balance between exploration and exploitation (using π_{random}). All the tests are made by letting MoGo play against GnuGo 3.6 with default mode. Komi are set to 7.5 points. The winning rates when MoGo plays black and white are given with the number of games played in each color (in parentheses). The number given after the ± is the standard error.

4.5.4 UCT with pruning

In this section we give heuristics to reduce the huge tree size especially in large Go board. The goal is to get a deeper local search in the tree by losing the global view of UCT. Obviously pruning heuristics may lead to a sub-optimal solution. First we define group by Go knowledge to reduce the branching factor in tree search. Then zone division is derived from group, which helps to have a more precise score. We use group and zone mode for 13×13 and 19×19 Go board. Figure 4.15 will give one example.

Remark 4 *As we are not very experienced for Go-knowledge-based programming and it is not the purpose of this work, we believe other programs like GnuGo and AyaGo, or Monte-Carlo programs have more clever pruning techniques. Some other techniques are mentioned in [Caz00][CH05]. However, our experimental results of combining UCT with pruning techniques are already encouraging.*

First we define one group as a set of strings and free intersections on a Go board according to certain Go knowledge, which gathers for example one big living group and its close enemies. We have implemented Common Fate Graph (CFG) [GGKH01] in our program to help the calculation of groups. The method starts from one string and recursively adds close empty intersections and strings close to these empty intersections until no more close strings are found within a distance controlled by a parameter.

In group mode, in the tree search part we search only the moves in the group instead of all over the Go board. In random simulation part there is no more such restriction. Using groups, we reduce the branching factor to less than 50 at the opening period. Then, depth of MoGo's tree could be around 7-8 on large Go board. Table 4.7 shows MoGo becomes competitive on 13×13 Go board by using group pruning technique. However, sophisticated pruning techniques are undoubtedly necessary to improve the level of Computer-Go programs.

Opponents	Win. Rate for B. Games	Win. rate for W. Games	Total Win. Rate
No group vs GG 0	53.2%(216)	51.8% (216)	$52\% \pm 2.4\%$
No group vs GG 8	24.2%(300)	30% (300)	$27\% \pm 1.8\%$
group vs GG 0	67.5% (80)	61.2% (80)	$64.3\% \pm 3.7\%$
group vs GG 8	51.9% (160)	60% (160)	$56\% \pm 2.7\%$

Table 4.7: MoGo with 70000 simulations per move, on 13×13 Go board, using or not the group mode heuristic against GnuGo 3.6 level 0 (GG 0) or 8 (GG 8).

As explained above, group mode limits the selection of moves in the tree search part. It has however no restriction on the random simulation. As the accuracy of the simulations becomes lower as the game length increases, we tried to generate the random moves only in a certain zone instead of on the whole Go board. The zones were defined using the groups

presented above. However the success were limited especially after the opening, when the zones are very difficult to define precisely. Interesting future research directions could be to define properly zones to limit the simulations lengths.

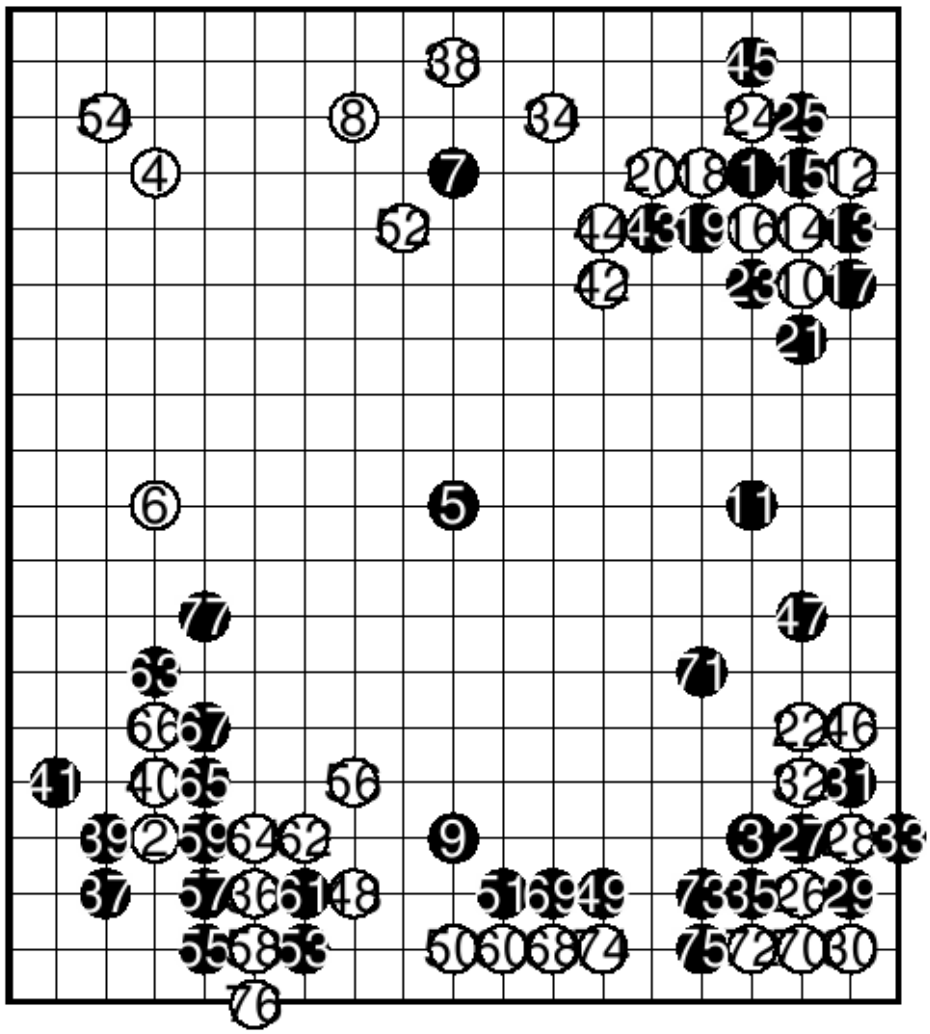

Figure 4.15: The opening of one game between MoGo and Indigo in the 18th KGS Computer Go Tournament. MoGo (Black) was in advantage at the beginning of the game, however it lost the game at the end.

4.5.5 First-play urgency

UCT works very well when the node is frequently visited as the trade-off between exploration and exploitation is well handled by UCB1 formula. However, for the nodes far from the root, whose number of simulations is very small, UCT tends to be too much exploratory. This is due to the fact that all the possible moves in one position are supposed to be explored before using the UCB1 formula. Thus, the values associated to moves in deep nodes are not meaningful, since the child-nodes of these nodes are not all explored yet and, sometimes even worse, the visited ones are selected in fixed order. This can lead to bad predicted sequences.

UCB1 algorithm begins by exploring each arm once, before using the formula (4.1). This can sometimes be inefficient especially if the number of trials is not large comparing to the number of arms. This is the case for numerous nodes in the tree (number of visits is small comparing to the number of moves). For example if an arm keeps returning 1 (win), there is no good reason to explore other arms. We have set a fixed constant named first-play urgency (FPU) in the algorithm. For each move, we name its urgency by the value of formula (4.1). The urgency value is set to the value of FPU (FPU is $+\infty$ by default) for each legal move before first visit (see line 15 in Table 4.2). Any node, after being visited at least once, has its urgency updated according to UCB1 formula. We play the move with the highest urgency. Thus, FPU $= +\infty$ ensures the exploration of each move once before further exploitation of any previously visited move. On the other way, smaller FPU ensures earlier exploitations if the first simulations lead to an urgency larger than FPU (in this case the other unvisited nodes are not selected). This improved the level of MoGo according to our experiment as shown in Table 4.8.

4.5.6 Rapid Action Value Estimation

The UCT algorithm must sample every action from a state $s \in \mathcal{T}$ before it has a basis on which to compare values. Furthermore, to produce a low-variance estimate of the value, each action in state s must be sampled multiple times. When the action space is large, this can cause slow learning. To solve this problem, we introduce a new algorithm UCT_{RAVE}, which forms a *rapid action value estimate* for action a in state s, and combines this online

FPU	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
1.4	37% ± 4.5% (100)	38% ± 5% (100)	37.5% ± 3.5%
1.2	46% ± 5% (100)	36% ± 5% (100)	41% ± 3.5%
1.1	45% ± 3% (250)	41% ± 3% (250)	43.4% ± 2.2%
1.0	49% ± 3% (300)	42% ± 3% (300)	45% ± 2%
0.9	47% ± 4% (150)	32% ± 4% (150)	40% ± 2.8%
0.8	40% ± 7% (50)	32% ± 6.5% (50)	36% ± 4.8%

Table 4.8: Influence of FPU (70000 simulations/move) (using π_{random}). All the tests are made by letting MoGo play against GnuGo 3.6 with default mode. Komi are set to 7.5 points. The winning rates when MoGo plays black and white are given with the number of games played in each color (in parentheses). The number given after the ± is the standard error.

knowledge into UCT.

Normally, Monte-Carlo methods estimate the value by averaging the return of all episodes in which a is selected immediately. Instead, we average the returns of all episodes in which a is selected at *any* subsequent time. In the domain of Computer Go, this idea is known as the *all-moves-as-first* heuristic [Bru93b]. However, the same idea can be applied in any domain where action sequences can be transposed.

Let $Q_{RAVE}(s,a)$ be the rapid value estimate for action a in state s. After each episode $s_1, a_1, s_2, a_2, ..., s_T$, the action values are updated for every state $s_{t_1} \in \mathcal{T}$ and every subsequent action a_{t_2} such that $a_{t_2} \in \mathcal{A}(s_{t_1})$, $t_1 \leq t_2$ and $\forall t < t_2, a_t \neq a_{t_2}$,

$$\begin{aligned} m(s_{t_1}, a_{t_2}) &\leftarrow m(s_{t_1}, a_{t_2}) + 1 \\ Q_{RAVE}(s_{t_1}, a_{t_2}) &\leftarrow Q_{RAVE}(s_{t_1}, a_{t_2}) \\ &+1/m(s_{t_1}, a_{t_2})[R_{t_1} - Q_{RAVE}(s_{t_1}, a_{t_2})] \end{aligned}$$

where $m(s,a)$ counts the number of times that action a has been selected at any time following state s.

The rapid action value estimate can quickly learn a low-variance value for each action. However, it may introduce some bias, as the value of an action usually depends on the exact

state in which it is selected. Hence we would like to use the rapid estimate initially, but use the original UCT estimate in the limit. To achieve this, we use a linear combination of the two estimates, with a decaying weight β,

$$\begin{aligned} Q^{\oplus}_{RAVE}(s,a) &= Q_{RAVE}(s,a) + c\sqrt{\frac{\log m(s)}{m(s,a)}} \\ \beta(s,a) &= \sqrt{\frac{k}{3n(s)+k}} \\ Q^{\oplus}_{UR}(s,a) &= \beta(s,a)Q^{\oplus}_{RAVE}(s,a) \\ &+ (1-\beta(s,a))Q^{\oplus}_{UCT}(s,a) \\ \pi_{UR}(s) &= \arg\max_a Q^{\oplus}_{UR}(s,a) \end{aligned}$$

where $m(s) = \sum_a m(s,a)$. The *equivalence parameter* k controls the number of episodes of experience when both estimates are given equal weight.

We tested the new algorithm $UCT_{RAVE}(\pi_{MoGo})$, using the default policy π_{MoGo}, for different settings of the equivalence parameter k. For each setting, we played a 2300 game match against GnuGo 3.7.10 (level 10). The results are shown in Figure 4.16, and compared to the $UCT(\pi_{MoGo})$ algorithm with 3000 simulations per move. The winning rate using UCT_{RAVE} varies between 50% and 60%, compared to 24% without rapid estimation. Maximum performance is achieved with an equivalence parameter of 1000 or more. This indicates that the rapid action value estimate is worth about the same as several thousand episodes of UCT simulation.

4.5.7 UCT with prior knowledge

The UCT algorithm estimates the value of each state by Monte-Carlo simulation. However, in many cases we have prior knowledge about the likely value of a state. We introduce a simple method to utilise offline knowledge, which increases the learning rate of UCT without biasing its asymptotic value estimates.

We modify UCT to incorporate an existing value function $Q_{prior}(s,a)$. When a new state and action (s,a) is added to the UCT representation $\mathcal{T}$, we initialise its value according

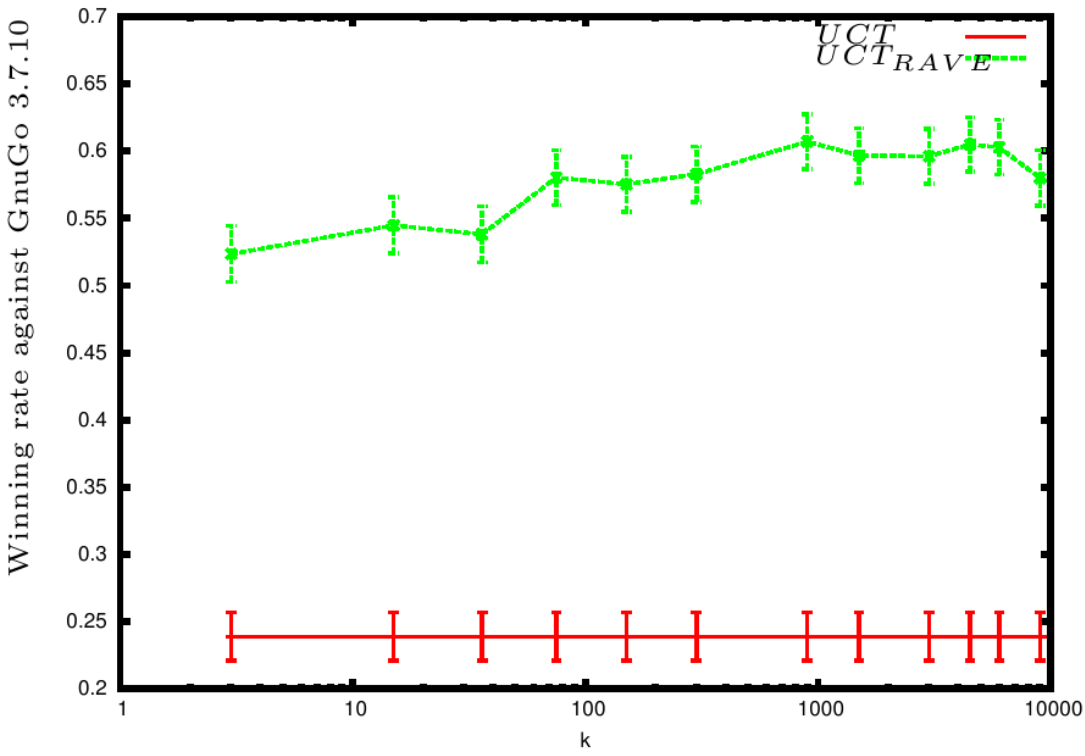

Figure 4.16: Winning rate of $UCT_{RAVE}(\pi_{MoGo})$ with 3000 simulations per move against GnuGo 3.7.10 (level 10), for different settings of the equivalence parameter k. The bars indicate twice the standard deviation. Each point of the plot is an average over 2300 complete games.

to our prior knowledge,

$$\begin{aligned} n(s,a) &\leftarrow n_{prior}(s_t,a_t) \\ Q_{UCT}(s_t,a_t) &\leftarrow Q_{prior}(s_t,a_t) \end{aligned}$$

The initial value n_{prior} estimates the *equivalent experience* contained in the prior value function. This indicates the number of episodes that UCT would require to achieve an estimate of similar accuracy. After initialisation, the value function is updated using the normal UCT update (see equations 4.2 and 4.3). We denote the new UCT algorithm using default policy π by $UCT(\pi, Q_{prior})$.

Algorithm	Wins .v. GnuGo
$UCT(\pi_{random})$	1.84 ± 0.22 % (3420)
$UCT(\pi_{MoGo})$	23.88 $\pm$0.85% (2520)
$UCT_{RAVE}(\pi_{MoGo})$	60.47 ± 0.79 % (3840)
$UCT_{RAVE}(\pi_{MoGo}, Q_{RLGO})$	69 ± 0.91 % (2600)

Table 4.9: Winning rate of the different UCT algorithms against GnuGo 3.7.10 (level 10), given 3000 simulations per move. The numbers after the $\pm$ correspond to the standard error. The total number of complete games is given in parentheses.

A similar modification can be made to the UCT_{RAVE} algorithm, by initialising the rapid estimates according to prior knowledge,

$$\begin{aligned} m(s,a) &\leftarrow m_{prior}(s_t, a_t) \\ Q_{RAVE}(s_t, a_t) &\leftarrow Q_{prior}(s_t, a_t) \end{aligned}$$

We compare several methods for generating prior knowledge in 9×9 Go. First, we use an *even-game* heuristic, $Q_{even}(s,a) = 0.5$, to indicate that most positions encountered on-policy are likely to be close. Second, we use a *grandfather* heuristic, $Q_{grand}(s_t, a) = Q_{UCT}(s_{t-2}, a)$, to indicate that the value with player P to play is usually similar to the value of the last state with P to play. Third, we use a handcrafted heuristic $Q_{MoGo}(s,a)$. This heuristic was designed such that greedy action selection would produce the best known default policy $\pi_{MoGo}(s,a)$. Finally, we use the linear combination of binary features, $Q_{RLGO}(s,a)$, learned offline by $TD(\lambda)$ (see section 4.3.6).

For each source of prior knowledge, we assign an equivalent experience $m_{prior}(s,a) = M_{eq}$, for various constant values of M_{eq}. We played 2300 games between $UCT_{RAVE}(\pi_{MoGo}, Q_{prior})$ and GnuGo 3.7.10 (level 10), alternating colours between each game. The UCT algorithms sampled 3000 episodes of experience at each move (see Figure 4.17), rather than a fixed time per move. In fact the algorithms have comparable execution time (Table 4.10).

The value function learned offline, Q_{RLGO}, outperforms all the other heuristics and increases the winning rate of the UCT_{RAVE} algorithm from 60% to 69%. Maximum performance is achieved using an equivalent experience of $M_{eq} = 50$, which indicates that Q_{RLGO}

Algorithm	Speed
$UCT(\pi_{random})$	6000 g/s
$UCT(\pi_{MoGo})$	4300 g/s
$UCT(\pi_{\varepsilon}), UCT(\pi_{\sigma}), UCT(\pi_{\tau})$	150 g/s
$UCT_{RAVE}(\pi_{MoGo})$	4300 g/s
$UCT_{RAVE}(\pi_{MoGo}, Q_{MoGo})$	4200 g/s
$UCT_{RAVE}(\pi_{MoGo}, Q_{RLGO})$	3600 g/s

Table 4.10: Number of simulations per second for each algorithm on a P4 3.4Ghz, at the start of the game. $UCT(\pi_{random})$ is faster but much weaker, even with the same time per move. Apart from $UCT(\pi_{RLGO})$, all the other algorithms have a comparable execution speed

is worth about as much as 50 episodes of UCT_{RAVE} simulation. It seems likely that these results could be further improved by varying the equivalent experience according to the variance of the prior value estimate.

4.5.8 Parallelization of UCT

The performance of UCT depends on the given time (equally the number of simulations) for each move. Tables 4.11 and 4.12 show its level improves as this number increases.

Seconds per move	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
5	26% ± 6% (50)	26% ± 6% (50)	26% ± 4.3%
20	41% ± 3% (250)	42% ± 3% (250)	41.8% ± 2.2%
60	53% ± 3.5% (200)	50% ± 3.5% (200)	51.5% ± 2.5%

Table 4.11: Uniform random mode and vanilla UCT with different times against GnuGo level 10.

We then modify UCT to make it work on a multi-processors machine with shared memory. The modifications to the algorithm are quite straightforward. All the processors share the same tree, and the access to the tree is locked by mutexes. As UCT is deterministic, all the threads could take exactly the same path in the tree, except for the leaf. The behavior of the multithreaded UCT as presented here is then different from the monothreaded UCT. Two experiments has then to be done. First, we can compare the level of MoGo

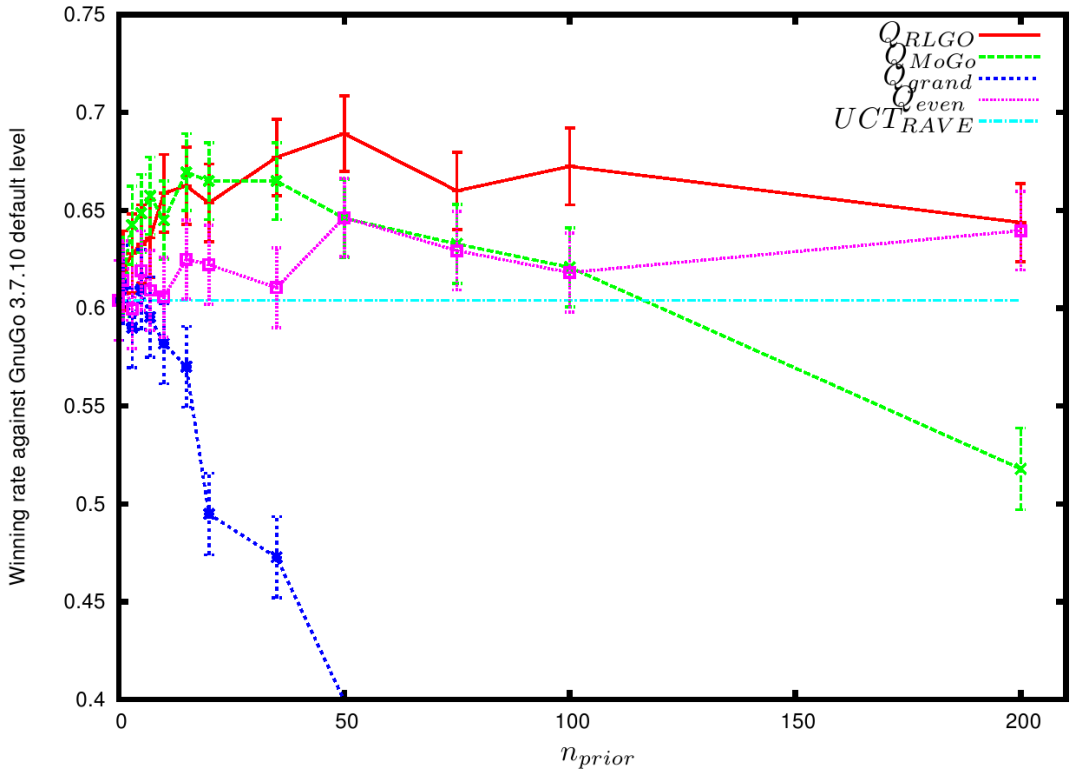

Figure 4.17: Winning rate of $UCT_{RAVE}(\pi_{MoGo})$ with 3000 simulations per move against GnuGo 3.7.10 (level 10), using different prior knowledge as initialisation. The bars indicate twice the standard deviation. Each point of the plot is an average over 2300 complete games.

using the monothreaded or the multithreaded algorithms while allowing the same number of simulations per move. All such experiments showed non significant differences in the play level[12]. Second, we can compare the level using the same time per move (the multithreaded version will then make more simulations per move). As UCT benefits from the computational power increase, the multithreaded UCT is efficient (+150 ELO on CGOS with 4 processors).

[12]we had only access to a 4 processors computer, the behavior can be very different with many more processors.

Simulations	Wins .v. GnuGo	CGOS rating
3000	69%	1960
10000	82%	2110
70000	92%	2320*

Table 4.12: Winning rate of $UCT_{RAVE}(\pi_{MoGo}, Q_{MoGo})$ against GnuGo 3.7.10 (level 10) when the number of simulations per move is increased. The asterisked version used on CGOS modifies the simulations/move according to the available time, from 300000 games in the opening to 20000. More complete results against other opponents and other board sizes can be found in Appendix C.

4.6 Conclusion and Perspectives

While most existing RL approaches in games are based on accurately learning the value function and using it in an alpha-beta search, this chapter has demonstrated the merits of combining a stochastic, robust, value function approximation, with the exploration vs exploitation strategy embedded in UCT [KS06].

The RL perspective is shifted, from accurately learning a good model (the exploitation of which is straightforward), to learning a model under bounded resources restrictions and using it as smartly as possible.

Along these lines, two contributions have been made. The first one is to show experimentally and explain theoretically that the qualities of the model and that of the strategy are not necessarily monotonically related. A second contribution relies on the exploitation of the domain knowledge, to improve the value function "deductively".

While the presented approach has achieved excellent results in the Game of Go (see Appendix C for more details), it is believed that it might address other application domains as well, specifically when the state space is large (frugal any-time algorithms are required) and the existing representation of the state space does not yet enable accurate value function learning.

Several research perspectives are opened by the presented work. Indeed, the approach can be parallelized. Early results show that a straightforward parallelization brings some performance gain and a cluster-compliant implementation of UCT would be a great step forward (incidentally, a good part of current Go programs now are based on UCT).

Secondly, it remains to draw the full consequences of the difference between value learning and policy optimization.

Thirdly, it is believed that both components of UCT and stochastic value process, are amenable to prior knowledge (e.g. using default policy outside of the current search tree, incorporating sophisticated patterns in the simulator, especially for end games, initializing the value function).

Chapter 5

Conclusions

Conclusion

This thesis presented some contributions in the field of Reinforcement Learning, or closely related fields.

Model learning

The first part (Chapter 2) presents a widely used model, namely Bayesian Networks (BNs). This graphical model framework can take advantage of the problem structure, e.g. representing Factored MDPs using Dynamic Bayesian Networks, giving opportunities to treat larger problems (scalability issue). The contributions concerning the BNs follow.

First, it is argued that the learning can be done through some L_2 loss function rather than the classical maximum likelihood. The first advantage is that the expectation approximation error using a model is directly linked to this L_2 measure of learning error. Second, optimization of this criterion with a wrong structure will lead to optimal parameters (given this structure), whereas frequentist approach will not. Third, this criterion naturally deals with hidden variables. However, this measure is not adapted to risk estimation (dealing with very small event probabilities) and is computationally much more expensive to compute. For the later point, we propose algorithms to compute this criterion efficiently and show that it is tractable.

Second, in this framework, learning error is theoretically bounded using covering numbers, a classical learning theory tool. From these theoretical bounds, we propose a new structural score, useful for structural learning in Bayesian Networks. In addition to the number of parameters, appearing in classical BN scores, our score measures also the entropy of the parameters distribution over the network. We empirically show its relevance.

Third, using the chosen criterion and the theoretical results, we propose learning algorithms with theoretical guarantees. A parameter learning algorithm, is shown to be Universal Consistent, and a structural learning algorithm guarantees the convergence to a parsimonious structure.

Future work may extend these theoretical results to other classes of models, including indirect graph models such as Markov Random Fields, because direct and indirect graph models do not encode the same conditional dependencies. In the partially observable case (hidden variables), our error bounds are loose and it would be useful to improve those bounds in this particular case. On the application side, the biggest issue is the computational cost of the learning, coming for the sum of squared probabilities over all possible assignments of variables (S term, see section 2.8). One of the most promising line of research is to adapt some of the very efficient Bayesian Networks inference algorithms to compute this sum in the learning criterion. Indeed, inference algorithms are designed to compute efficiently (taking advantage of the network structure) sum of probabilities (marginalization), while this term is a sum of squared probabilities. Those issues are known under the factor graphs and sum-product terminologies [AM97, KFL01].

Robust Stochastic Dynamic Programmings

The second part of this thesis work (Chapter 3) deals with Stochastic Dynamic Programming (SDP) in a continuous state space and action space. The continuous case brings issues which do not exist in the (small) discrete case, namely function approximation (regression), combined with sampling, and the optimization step in SDP, i.e. computation of the argmax over the actions at one time step.

We propose an empirical study effect of non linear optimization algorithms, regression

learning and sampling on the SDP framework[1]. On the non linear optimization issue, it is shown that robustness (in the sense of avoiding local minima, dealing with unsmooth functions, dealing with noise, ...), is the most valuable property. Evolutionary algorithms, based on a population of points instead of only one point for algorithms like BFGS, give the best results.

In the regression problem, other robustness issues arise, the most important being the probability of appearance of "false" optima in the learned value function, i.e. a minimum which does not correspond to a minimum in the true value function. On our benchmarks, the SVM algorithm gives the most stable result, beating many other classical regression algorithms for the control tasks.

The third contribution concerns sampling, which is of main importance in SDP, as a good sampling can reduce the number of examples used to learn the function, and each example represents a high computational cost. The chapter presents some theoretical insights about sampling, showing that a minimum amount of randomness is necessary to guarantee consistency, while the amount of randomness can be very small, allowing derandomization to make efficient sampling. A new sampling method in SDP is proposed, based on an evolutionary algorithm, and can be used with other improved sampling methods in Dynamic Programming. Empirical evidences show that derandomized blind sampling works best for the benchmarked control tasks, while performances of non blind samplings greatly vary depending on the task.

An immediate perspective of those studies would be to tackle a real world control problem which needs advanced regression/optimization techniques due to the scale of the problem. For example the domain of energy production would be a good real scale testbed, with great both economic and ecological impacts, where heuristics are used to reduce the problem to smaller dimensions. Our OpenDP framework needs further developments to reach a mature level, notably on the ease of use. The empirical analysis could be extended with the integration of other classical extension of SDP to get the best of state of the art performance, in order to get a real world application.

[1] Those studies have been performed in our framework, OpenDP, which can be used by the community as a tool to compare algorithms in different benchmark problems. This framework is made Open Source so that every researcher or student can used part of the code, and contribute by his algorithm or new tools.

Computer-Go

The third part (Chapter 4) deals with a high dimensional control problem, the game of Go. This problem is a great challenge for the Machine Learning field, with many difficult requirements. We believe that the methods presented in this chapter can also be applied in other challenging discrete control problems, making the game of Go a testbed for developing those techniques. The contributions of this chapter lie around the UCT algorithm, from its application to computer go, to improvements in both the tree search and in the Monte-Carlo simulation. Those contributions have been applied in our program MoGo[2].

The role of the simulation policy is to estimate the value of a state (a position). The first contribution concerns the introduction of "sequence-like simulations", where forced sequences emerge from the use of local patterns around the previous move. The improvement of the UCT algorithm using this simulation policy is very significant along with the own strength of the simulation policy. However, we propose empirical evidences showing that the strength of the UCT algorithm is not monotonous in the strength of the simulation policy. We also propose some theoretical insights on this surprising result.

The second set of contributions concern more specifically the tree search part of the algorithm. The most significant improvement uses a simple yet efficient generalization of actions (moves) values over the tree. This generalization improves the play level in small boards, but more interestingly enables scalability of the algorithm on bigger boards (corresponding to larger state and action spaces).

Future work

We have presented both a model learning and a decision-making module (using a UCT-like algorithm) in this thesis. A natural extension would be to combine both of these modules together in a real world application. Another challenging testbed problem is the game of poker, where, contrary to the game of Go, a model of the opponent is necessary. The uncertainty on the opponent represents the uncertainty on the transition function of the underlying MDP (the rules of the game are obviously known). One on-going work in our group

[2]MoGo won several international tournaments in all board sizes, along with some results against strong humans in smaller board sizes, and is the world strongest Go program at the time of writing

combines the learning criterion presented in Chapter 2 with a Monte-Carlo tree search algorithm in Texas Hold'em, limit, two players. The opponent model can be represented as a Bayesian Network, as a tool to approximate the probability of each action given observable and hidden variables. The criterion we proposed (in Chapter 2) is appropriate here, as the profit expectation is what matters in poker, and a simple online extension of the proposed algorithm would be suitable. This opponent model can then be used in place of a self-play simulation policy for a UCT based algorithm along the lines of Chapter 4.

As real world problems have some underlying structure, one of the most promising lines of research for UCT-based tree search algorithms is to generalize between nodes in the tree. Currently UCT is a tabular algorithm, treating each position as a totally new problem. Our RAVE algorithm puts some generalization into the algorithm by sharing action values between the nodes. It achieves a very significant improvement, while remaining very simple. Further generalization, i.e. taking advantage of the problem structure, can be brought using a (simple and fast) function approximation, like the one used in the game of Go by [SS07]. The idea is to efficiently learn online an approximation of the state values but specialized on the state distribution induced by UCT. Hence, the approximation is made on a (tiny) subset of all possible states, making a better use of the approximation capabilities of the function approximator than trying to approximate the whole state space. This is also related to cost sensitive learning where each learning error does not have the same cost: it is pointless (no associated cost) to approximate the value of a state which is hardly reached, and it would be better to keep the approximation capabilities for frequently reached states (where the cost of an error is large).

Another line of research is to tackle the algorithmic issues of UCT in a massive parallel framework. Current evolution of hardware emphasizes multicore processors, and clusters of machines. While straightforward implementations of UCT in shared memory machines have been successful, theoretical studies on the behavior of UCT on a cluster of machines (facing heavy[3] communication costs) would benefit large scale applications. The generalization between different states would also be beneficial in a massive parallel framework, enabling a more structured clustering of the computations to be spread over

[3]Heavy costs compared to the cost of a memory access, and also compared to the computation of one evaluation of a state.

the network, and allowing a more efficient (lighter) information sharing.

This thesis has emphasized the interactions of Reinforcement Learning with other research goals, such as supervised learning, unsupervised learning, discrete and real value optimization, numerical integration and learning theory. We have also emphasized the importance of real world applications and explored how simple algorithms can be scaled up to large problems. We believe that all of these elements can be integrated together seamlessly, and that the performance of such a system would be greater than the sum of its parts.

Appendix A

Some OpenDP screenshots

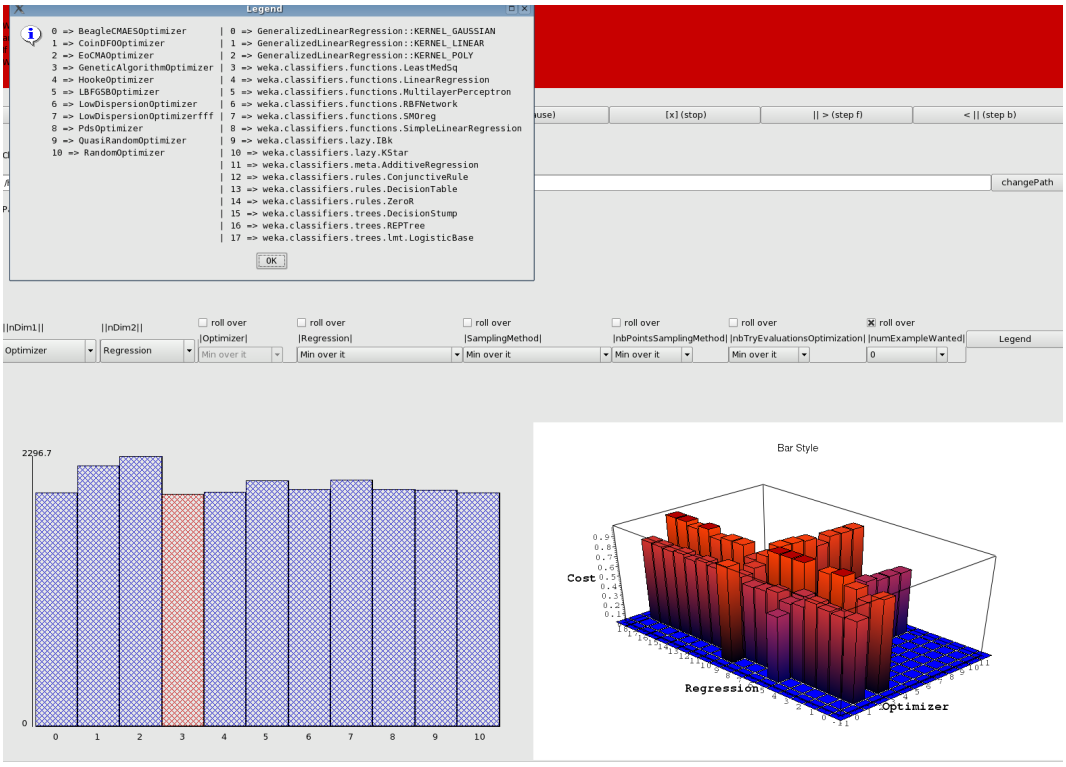

Figure A.1: Screenshot of the ”analyser” module of OpenDP, enabling the visual comparison of the algorithm performances after a set of experiments.

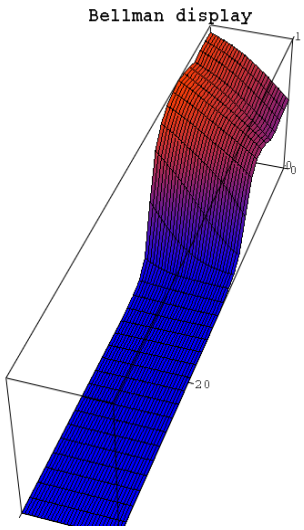

Figure A.2: The Bellman value function for the stock management problem (enabling rescaling, zoom, rotations).

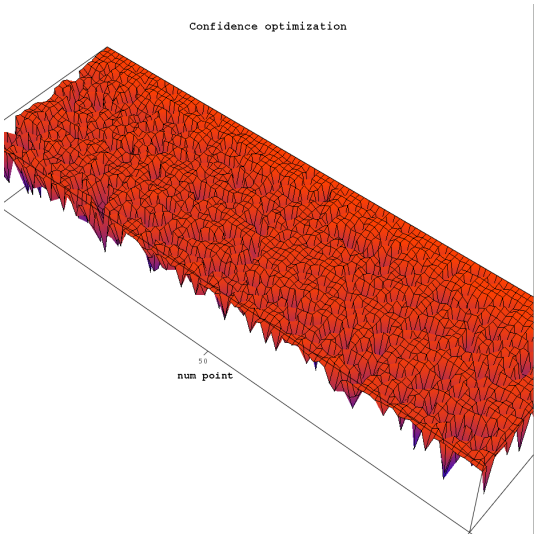

Figure A.3: Veall Test [MR90] based on the asymptotic confidence interval for the first order statistics computed by De Haan [dH81]. For each sampled point, is plotted the confidence for the optimization step to be close to the optimum.

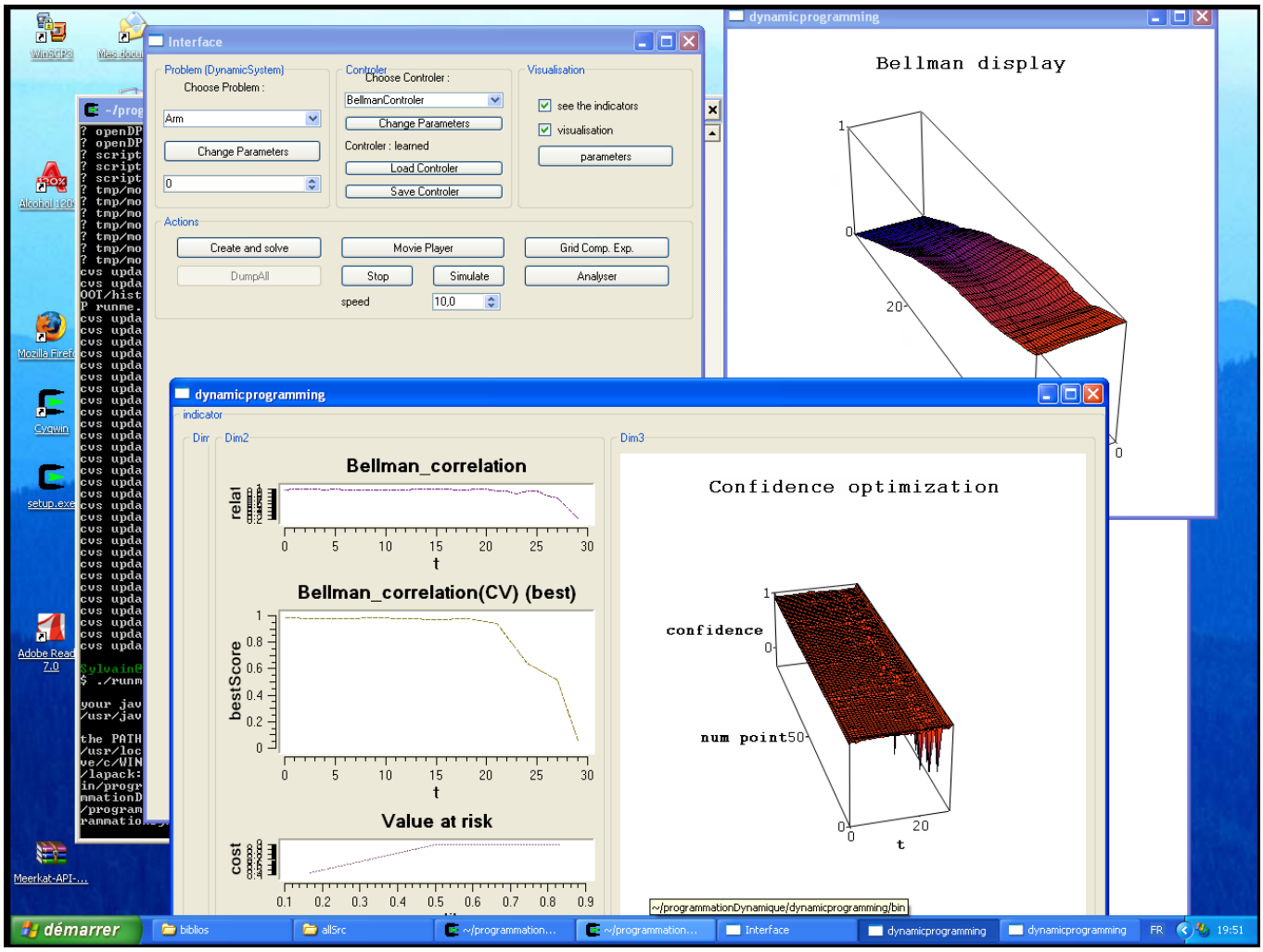

Figure A.4: OpenDP on windows XP: Main interface (top left), the Bellman value function (top right), and SDP online indicators, including correlation (bottom left) and confidence optimization (bottom right).

Appendix B

Experimental Results of Chapter 3

B.1 Description of algorithm "EA"

This algorithm involves a Gaussian isotropic mutation with standard deviation $\frac{\sigma}{\sqrt[d]{n}}$ with n the population size d the search space dimension. The crossover between two individuals x and y gives birth to two individuals $\frac{1}{3}x+\frac{2}{3}y$ and $\frac{2}{3}x+\frac{1}{3}y$. Let $\lambda_1,\lambda_2,\lambda_3,\lambda_4$ be such that $\lambda_1+\lambda_2+\lambda_3+\lambda_4=1$; we define S_1 the set of the $\lambda_1.n$ best individuals, S_2 the $\lambda_2.n$ best individuals among the others. At each generation, the new offspring is (i) a copy of S_1 (ii) $n\lambda_2$ cross-overs between individuals from S_1 and individuals from S_2 (iii) $n\lambda_3$ mutated copies of individuals from S_1 (iv) $n\lambda_4$ individuals randomly drawn uniformly in the domain.

- we copy the $n\lambda_1$ best individuals (set S_1).
- we combine the $n\lambda_2$ following best individuals with the individuals of S_1 (rotating among S_1 if $\lambda_1<\lambda_2$).
- we mutate $n\lambda_3$ individuals among S_1 (again rotating among S_1 if $\lambda_1<\lambda_3$).
- we randomly generate $n\times\lambda_4$ other individuals, uniformly on the domain.

The parameters are $\sigma=0.08,\lambda_1=1/10,\lambda_2=2/10,\lambda_3=3/10,\lambda_4=4/10$; the population size is the square-root of the number of fitness-evaluations allowed. These parameters are standard ones from the library.

B.2 Non Linear Optimization results

See tables A.1 to A.6.

Stock Management Baseline Dimension x1 → 4			
Optimizer	score	Std	time (s)
LowDispersionfff	2815.99	±4.91	41.68
EoCMA	2927.43	±6.64	55.96
QuasiRandom	3005	±9.38	38.98
RestartLBFGSB	3014.25	±8.12	76.87
LowDispersion	3015.21	±7.67	41.24
EA	3016.29	±7.42	41.88
Random	3016.75	±7.87	36.77
LBFGSB	3019.34	±6.60	27.30
RestartHooke	3019.99	±6.02	40.55
EANoMem	3023.89	±6.08	41.92
Hooke	3024.45	±4.66	40.20

Stock Management Baseline Dimension x2 → 8			
Optimizer	score	Std	time (s)
EoCMA	2638.18	±6.43	43.90
LowDispersionfff	2700.61	±1.61	36.56
QuasiRandom	3016.49	±1.80	34.65
Random	3022.1	±1.80	32.00
EA	3031.63	±1.78	37.83
EANoMem	3031.63	±1.78	38.49
Hooke	3031.63	±1.78	37.52
LBFGSB	3031.63	±1.78	7.21
LowDispersion	3031.63	±1.78	37.17
RestartHooke	3031.63	±1.78	37.76
RestartLBFGSB	3031.63	±1.78	36.28

Stock Management Baseline Dimension x3 → 12			
Optimizer	score	Std	time (s)
EoCMA	2627.31	±9.47	48.83
LowDispersionfff	2631.58	±1.60	40.32
QuasiRandom	3018.07	±1.79	37.63
Random	3031.37	±1.78	35.30
EA	3031.63	±1.78	40.70
EANoMem	3031.63	±1.78	41.85
Hooke	3031.63	±1.78	42.58
LBFGSB	3031.63	±1.78	8.94
LowDispersion	3031.63	±1.78	40.11
RestartHooke	3031.63	±1.78	42.75
RestartLBFGSB	3031.63	±1.78	41.87

Stock Management Baseline Dimension x4 → 16			
Optimizer	score	Std	time (s)
EoCMA	2678.46	±9.00	60.89
LowDispersionfff	2741.09	±1.60	48.51
Random	3015.42	±1.79	41.75
QuasiRandom	3025.71	±1.78	46.30
EA	3031.63	±1.78	50.65
EANoMem	3031.63	±1.78	53.35
Hooke	3031.63	±1.78	58.74
LBFGSB	3031.63	±1.78	11.20
LowDispersion	3031.63	±1.78	49.46
RestartHooke	3031.63	±1.78	57.16
RestartLBFGSB	3031.63	±1.78	45.02

Table B.1: Results on the Stock Management problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ or $\times 4$ which give dimension 4, 8, 12 or 16). It represents the dimension of the action space. The dimension of the state space increases accordingly. The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. CMA algorithm and LowDispersionfff are the two best ranked optimizers, beating by far the others. CMA is best on 3 over 4 dimension sizes.

Stock Management V2 Baseline Dimension x1 → 4			
Optimizer	score	Std	time (s)
LowDispersion	354.92	±7.26	22.21
EANoMem	402.30	±7.25	22.37
RestartLBFGSB	435.93	±8.17	49.98
EA	449.39	±6.18	22.06
QuasiRandom	452.07	±6.29	20.90
LowDispersionfff	464.93	±5.55	22.18
Random	524.1	±6.74	19.48
EoCMA	657.78	±12.38	32.96
Hooke	933.12	±12.17	19.59
RestartHooke	939.14	±12.54	27.03
LBFGSB	994.37	±16.02	11.46

Stock Management V2 Baseline Dimension x2 → 8			
Optimizer	score	Std	time (s)
EoCMA	255.32	±8.32	36.13
RestartHooke	270.12	±9.59	25.04
LBFGSB	270.65	±6.45	13.42
EA	273.82	±11.12	25.89
LowDispersion	274.98	±9.05	26.34
Random	275.18	±9.26	23.16
RestartLBFGSB	278.96	±7.75	46.96
Hooke	279.06	±7.11	24.99
EANoMem	280.91	±9.23	26.74
LowDispersionfff	280.93	±8.58	26.39
QuasiRandom	296.58	±6.40	25.10

Table B.2: Results on the Stock Management problem, second version (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$ or $\times 2$ which give dimension 4 or 8). It represents the dimension of the action space. The dimension of the state space increases accordingly. The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. With low dimension (left), LowDispersion optimizer gives the best result, with a very significant difference. In larger dimension (right), algorithms are closer, while CMA becomes the best (as in the first version of the Stock and Demand problem).

B.3 Learning results

See tables A.7 to A.18.

Arm Baseline Dimension x1 → 3			
Optimizer	score	Std	time (s)
LowDispersion	11.15	±0.23	56.61
RestartLBFGSB	11.43	±0.27	118.04
RestartHooke	11.48	±0.28	55.57
LBFGSB	11.53	±0.29	20.53
Hooke	11.66	±0.30	55.24
EANoMem	11.82	±0.26	57.51
EA	12.04	±0.26	57.04
QuasiRandom	12.47	±0.26	54.49
Random	12.90	±0.24	52.78
LowDispersionfff	13.92	±0.22	56.52
EoCMA	17.73	±0.12	63.96

Arm Baseline Dimension x2 → 6			
Optimizer	score	Std	time (s)
Hooke	10.43	±0.26	61.16
RestartHooke	10.53	±0.28	61.29
RestartLBFGSB	10.58	±0.26	115.20
LBFGSB	11.04	±0.27	46.53
LowDispersion	11.08	±0.25	60.96
EANoMem	11.14	±0.22	62.20
EA	12.24	±0.22	61.07
QuasiRandom	12.44	±0.22	59.14
Random	12.98	±0.21	56.36
LowDispersionfff	15.12	±0.19	60.58
EoCMA	18.89	±0.08	65.24

Arm Baseline Dimension x3 → 9			
Optimizer	score	Std	time (s)
LowDispersion	11.03	±0.21	65.78
Hooke	11.05	±0.21	66.65
RestartHooke	11.17	±0.21	66.85
RestartLBFGSB	11.47	±0.22	138.16
LBFGSB	11.86	±0.20	107.83
EANoMem	12.27	±0.20	67.65
QuasiRandom	12.84	±0.19	63.22
EA	12.88	±0.21	66.03
Random	13.29	±0.21	60.30
LowDispersionfff	15.06	±0.17	65.17
EoCMA	18.70	±0.08	74.10

Arm Baseline Dimension x4 → 12			
Optimizer	score	Std	time (s)
LowDispersion	10.64	±0.19	69.96
Hooke	11.13	±0.22	77.65
RestartLBFGSB	11.38	±0.22	208.61
RestartHooke	11.39	±0.20	77.96
EA	11.81	±0.19	70.75
LBFGSB	11.85	±0.19	196.40
QuasiRandom	12.28	±0.18	67.91
EANoMem	12.30	±0.19	72.27
EA	12.39	±0.17	64.67
LowDispersionfff	15.49	±0.16	69.49
EoCMA	18.60	±0.08	80.51

Table B.3: Results on the ”Arm” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ or $\times 4$ which give dimension 3, 6, 9 or 12). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. LowDispersion and Hooke optimizers are the best while LowDispersionfff and CMA are the worst by far. This problem is typically a ”bang-bang” problem, where the optimal action is on the frontier of the action space. LowDispersionfff and CMA which evolve far from the frontiers are very bad.

Fast Obstacle Avoidance Baseline Dimension x1 → 1			
Optimizer	score	Std	time (s)
RestartHooke	584.84	±8.70	30.78
LowDispersion	617.42	±11.39	22.56
QuasiRandom	626.51	±16.34	21.26
EA	638.63	±15.56	22.68
EANoMem	643.18	±16.58	22.94
LowDispersionfff	643.18	±15.52	22.6
Random	649.24	±16.28	20.11
RestartLBFGSB	702.27	±16.13	116.58
EoCMA	821.21	±18.88	40.77
Hooke	968.18	±12.89	12.81
LBFGSB	1003.79	±1.96	12.87

Many-obstacles avoidance Baseline Dimension x1 → 1			
Optimizer	score	Std	time (s)
EA	72.95	±1.37	21.57
LowDispersion	72.95	±1.44	21.52
RestartLBFGSB	73.03	±1.37	74.57
EANoMem	73.33	±1.31	21.88
LowDispersionfff	73.48	±1.36	21.44
QuasiRandom	74.01	±1.45	20.18
Random	74.62	±1.36	19.00
RestartHooke	77.42	±1.60	29.60
EoCMA	85.45	±1.57	37.74
Hooke	100	±1.e-16	12.35
LBFGSB	100	±1.e-16	12.75

Table B.4: Results on the ”fast obstacle avoidance” problem (left) and the ”many-obstacles avoidance” problem (right) (see section 3.2.2 for details). These two problems has a very low dimension (1). Hooke and LBFGSB, which are local optimizers perform badly (in the second problem they lead to the worst possible cost). Making them global optimizers using a restart (start again from a new random point, each time a local convergence happens), is very efficient, RestartHooke even becomes the best on the avoidanceproblem. On the two problem, the differences between the top optimizers are small.

Many bots Baseline Dimension x1 → 4			
Optimizer	score	Std	time (s)
EA	218.93	±2.07	46.94
EANoMem	225.15	±2.24	47.34
Random	241.36	±2.68	44.68
QuasiRandom	242.27	±2.40	46.29
LowDispersion	250.60	±2.72	47.97
EoCMA	277.57	±4.02	82.79
LowDispersionfff	314.69	±8.23	47.85
RestartHooke	331.66	±12.76	46.29
Hooke	341.06	±12.23	46.16
LBFGSB	447.57	±5.70	69.33
RestartLBFGSB	600	±1.e-16	109.64

Many bots Baseline Dimension x2 → 8			
Optimizer	score	Std	time (s)
EANoMem	836.21	±7.32	58.82
EA	856.06	±8.39	57.78
LowDispersionfff	915.15	±11.05	61.70
QuasiRandom	936.51	±6.46	56.05
Random	940.30	±6.67	54.07
LowDispersion	962.12	±12.37	59.38
EoCMA	1023.03	±12.64	98.08
Hooke	1025.91	±28.59	60.84
RestartHooke	1049.09	±25.94	61.20
LBFGSB	1175.3	±9.24	202.43
RestartLBFGSB	1395.15	±3.73	237.60

Many bots Baseline Dimension x3 → 12			
Optimizer	score	Std	time (s)
LowDispersionfff	1505.61	±16.15	76.10
EANoMem	1564.24	±11.24	77.20
EA	1624.55	±11.47	69.12
QuasiRandom	1673.48	±9.11	75.09
Random	1711.21	±11.08	63.95
RestartHooke	1776.67	±30.60	107.04
Hooke	1780.3	±31.30	80.26
EoCMA	1823.79	±13.20	123.48
LowDispersion	1844.09	±13.40	102.82
LBFGSB	1925.15	±12.52	430.21
RestartLBFGSB	2200	±1.e-16	478.96

Many bots Baseline Dimension x4 → 16			
Optimizer	score	Std	time (s)
EANoMem	2361.36	±9.52	86.95
EA	2382.73	±12.04	85.80
LowDispersionfff	2435.3	±21.04	84.27
QuasiRandom	2441.06	±10.52	83.00
LowDispersion	2466.97	±16.48	82.66
Random	2476.97	±11.06	96.19
EoCMA	2539.85	±21.71	144.84
RestartHooke	2591.06	±34.71	97.47
Hooke	2650.15	±34.37	96.25
LBFGSB	2697.27	±13.04	736.73
RestartLBFGSB	3001.21	±0.55	869.19

Table B.5: Results on the "Many bots" problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 4, 8, 12 or 16). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. Gradient based algorithms as LBFGSB and its "restart" version perfom very badly, being the worst in all dimensions. The simple evolutionary algorithm called "EA" perform well, which reveals the very unsmooth properties of the optimization.

Away Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
LowDispersion	1.55	±0.22	55.90
Hooke	1.59	±0.20	45.81
LBFGSB	1.67	±0.20	15.42
RestartHooke	1.70	±0.21	72.57
EA	1.73	±0.20	55.82
EANoMem	1.74	±0.23	58.55
QuasiRandom	1.80	±0.22	52.91
RestartLBFGSB	1.84	±0.24	159.39
Random	2.03	±0.25	53.16
LowDispersionfff	2.19	±0.29	56.06
EoCMA	3.03	±0.38	71.09

Away Baseline Dimension x2 → 4			
Optimizer	score	Std	time (s)
LowDispersion	1.05	±0.17	74.55
LBFGSB	1.22	±0.19	29.44
RestartLBFGSB	1.25	±0.18	170.67
Hooke	1.26	±0.19	72.48
EA	1.32	±0.20	75.63
RestartHooke	1.34	±0.22	72.78
EANoMem	1.41	±0.28	76.53
QuasiRandom	1.62	±0.27	72.44
Random	1.78	±0.29	70.16
LowDispersionfff	2.86	±0.48	76.25
EoCMA	7.51	±0.77	81.89

Away Baseline Dimension x3 → 6			
Optimizer	score	Std	time (s)
LowDispersion	0.74	±0.15	86.36
EANoMem	0.75	±0.14	88.58
LBFGSB	0.85	±0.19	51.41
RestartLBFGSB	0.87	±0.17	165.76
Hooke	0.93	±0.19	87.19
RestartHooke	0.94	±0.19	87.46
EA	0.97	±0.19	87.76
QuasiRandom	1.23	±0.23	84.25
Random	1.27	±0.25	82.01
LowDispersionfff	3.12	±0.47	90.03
EoCMA	6.71	±0.70	90.92

Away Baseline Dimension x4 → 8			
Optimizer	score	Std	time (s)
LowDispersion	0.31	±0.07	90.41
Hooke	0.62	±0.16	95.02
EANoMem	0.65	±0.14	95.38
RestartHooke	0.66	±0.15	96.38
LBFGSB	0.70	±0.15	75.54
RestartLBFGSB	0.70	±0.15	162.18
EA	0.76	±0.19	93.77
QuasiRandom	0.76	±0.15	89.55
Random	0.85	±0.18	85.35
LowDispersionfff	2.98	±0.44	98.72
EoCMA	5.28	±0.63	96.92

Table B.6: Results on the "Away" problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$, $\times 4$ which give dimension 2, 4, 6 or 8). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. Blind optimizer as LowDispersion optimizer is best for each dimension, while the difference is not significant compared to the first bests optimizers. Results are similar from results for the "Arm" problem (see Table B.3). This problem is typically a "bang-bang" problem, where the optimal action is on the frontier of the action space. LowDispersionfff and CMA which evolve far from the frontiers are very bad.

Stock Management (300 points) Baseline Dimension x1 → 4			
Optimizer	score	Std	time (s)
SVMLap	1866.87	± 1.67	36.20
SVMGauss	1867.52	± 1.68	35.93
SVMGaussHP	2147.91	± 24.96	119.50
IBk	2411.20	± 3.82	207.78
LinearR	2455.74	± 1.79	36.62
SimpleLinearR	2457.70	± 1.76	32.48
LRK(linear)	2741.17	± 5.83	71.49
DecTable	2802.88	± 2.61	43.54
LRK(gaus)	2931.81	± 5.13	128.56
AR	2935.64	± 5.96	32.75
LRK(poly)	2944.80	± 7.31	120.93
RBFNetwork	3029.99	± 1.78	54.72
DecisionStump	3030.78	± 1.73	31.89
MLP	3031.19	± 0	48.70
LogisticBase	3031.24	± 1.72	254.97
LeastMedSq	3031.27	± 0	408.20
REPTree	3031.81	± 0	33.23

Stock Management (300 points) Baseline Dimension x2 → 8			
Optimizer	score	Std	time (s)
IBk	2872.95	± 3.49	304.80
SVMGaussHP	2989.18	± 6.11	90.19
SVMGauss	2995.96	± 1.55	39.71
SVMLap	2996.22	± 1.55	40.03
REPTree	3017.43	± 2.14	39.05
LinearR	3026.21	± 1.69	43.01
AR	3028.39	± 1.80	39.03
RBFNetwork	3029.12	± 1.72	62.60
DecTable	3030.53	± 1.74	50.59
DecisionStump	3031.27	± 0	37.66
LeastMedSq	3031.27	± 0	313.49
LogisticBase	3031.27	± 0	203.39
LRK(gaus)	3031.27	± 0	164.37
LRK(linear)	3031.27	± 0	79.97
LRK(poly)	3031.27	± 0	109.38
MLP	3031.27	± 0	40.04
SimpleLinearR	3031.27	± 0	37.01

Stock Management (300 points) Baseline Dimension x3 → 12			
Optimizer	score	Std	time (s)
AR	3031.27	± 0	40.30
DecisionStump	3031.27	± 0	39.91
DecTable	3031.27	± 0	52.47
IBk	3031.27	± 0	427.59
LeastMedSq	3031.27	± 0	296.13
LinearR	3031.27	± 0	45.93
LogisticBase	3031.27	± 0	169.31
LRK(gaus)	3031.27	± 0	175.45
LRK(linear)	3031.27	± 0	89.09
LRK(poly)	3031.27	± 0	108.20
MLP	3031.27	± 0	37.01
RBFNetwork	3031.27	± 0	70.58
REPTree	3031.27	± 0	42.40
SimpleLinearR	3031.27	± 0	39.28
SVMLap	3031.27	± 0	33.00
SVMGauss	3031.27	± 0	32.76
SVMGaussHP	3031.27	± 0	42.84

Stock Management (300 points) Baseline Dimension x4 → 16			
Optimizer	score	Std	time (s)
AR	3031.27	± 0	41.72
DecisionStump	3031.27	± 0	42.30
DecTable	3031.27	± 0	55.37
IBk	3031.27	± 0	531.85
LeastMedSq	3031.27	± 0	307.50
LinearR	3031.27	± 0	48.41
LogisticBase	3031.27	± 0	159.23
LRK(gaus)	3031.27	± 0	189.14
LRK(linear)	3031.27	± 0	101.80
LRK(poly)	3031.27	± 0	121.47
MLP	3031.27	± 0	47.74
RBFNetwork	3031.27	± 0	83.14
REPTree	3031.27	± 0	49.15
SimpleLinearR	3031.27	± 0	43.00
SVMLap	3031.27	± 0	38.10
SVMGauss	3031.27	± 0	36.07
SVMGaussHP	3031.27	± 0	52.04

Table B.7: Results on the ”Stock Management” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 4, 8, 12 or 16). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Stock Management V2 (300 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
SVMGaussHP	366.45	± 7.20	96.31
SVMLap	409.54	± 7.33	18.03
SVMGauss	418.11	± 7.32	17.97
IBk	535.29	± 8.42	106.03
LinearR	696.87	± 4.28	21.40
REPTree	1127.76	± 16.30	20.15
SimpleLinearR	1226.58	± 2.39	19.68
AR	1288.96	± 10.45	19.32
DecTable	1304.35	± 26.28	24.24
LRK(gaus)	1442.61	± 17.93	71.73
LRK(linear)	1456.01	± 11.73	38.30
DecisionStump	1593.02	± 6.99	19.34
LeastMedSq	1649.76	± 2.04	189.40
LogisticBase	1656.80	± 1.56	93.94
MLP	1657.01	± 1.54	19.00
RBFNetwork	1664.85	± 2.02	31.00
LRK(poly)	1676.11	± 4.42	70.01

Stock Management V2 (300 points) Baseline Dimension x2 → 4			
Optimizer	score	Std	time (s)
SimpleLinearR	251.91	± 3.87	20.66
LinearR	273.08	± 6.00	22.84
LeastMedSq	277.36	± 4.93	179.97
SVMGauss	278.19	± 10.28	20.54
SVMGaussHP	281.54	± 6.69	64.48
SVMLap	282.26	± 7.83	20.80
LRK(linear)	291.22	± 4.60	42.24
IBk	292.79	± 5.29	146.75
MLP	299.13	± 3.44	19.64
LogisticBase	299.47	± 3.20	117.21
AR	300.90	± 3.40	20.13
LRK(poly)	304.13	± 4.40	64.92
REPTree	306.59	± 3.11	22.06
LRK(gaus)	306.78	± 3.55	90.67
DecisionStump	307.64	± 2.89	20.33
RBFNetwork	309.37	± 3.80	34.17
DecTable	314.74	± 2.68	26.40

Table B.8: Results on the ”Stock Management V2” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$ which give dimension 2 and 4). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Fast Obstacle Avoidance (300 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
IBk	550.00	± 0.00	56.60
SVMLap	626.43	± 13.26	18.21
SVMGauss	627.86	± 14.01	18.13
DecTable	685.71	± 6.99	20.64
REPTree	737.86	± 15.84	16.07
MLP	758.57	± 24.46	17.87
SVMGaussHP	927.86	± 15.18	53.94
LRK(linear)	993.57	± 6.43	48.16
AR	997.14	± 2.01	15.85
DecisionStump	1000.00	± 0.00	15.43
LeastMedSq	1000.00	± 0.00	204.43
LinearR	1000.00	± 0.00	17.56
LogisticBase	1000.00	± 0.00	48.45
LRK(gaus)	1000.00	± 0.00	58.62
RBFNetwork	1000.00	± 0.00	26.06
SimpleLinearR	1000.00	± 0.00	15.60
LRK(poly)	1016.43	± 7.17	36.53

Many-obstacle avoidance (300 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
LRK(poly)	60.36	± 0.16	36.73
MLP	60.79	± 0.22	18.13
IBk	64.79	± 0.39	57.24
SVMLap	71.29	± 1.19	17.53
SVMGauss	73.57	± 1.40	17.48
SVMGaussHP	79.64	± 2.33	118.87
REPTree	92.50	± 1.07	16.77
AR	93.36	± 0.92	17.01
LeastMedSq	96.79	± 1.20	200.73
DecisionStump	99.29	± 0.31	16.35
SimpleLinearR	99.36	± 0.38	16.42
DecTable	99.64	± 0.29	20.79
LRK(linear)	99.64	± 0.29	48.58
LinearR	100.00	± 0.00	18.16
LogisticBase	100.00	± 0.00	39.73
LRK(gaus)	100.00	± 0.00	58.32
RBFNetwork	100.00	± 0.00	25.73

Table B.9: Results on the ”Fast obstacle avoidance” (left) and ”Many-obstacle avoidance” (right) problems (see section 3.2.2 for details). These are in dimension 2. See table 3.3 for a summary.

Many bots (300 points) Baseline Dimension x1 → 8			
Optimizer	score	Std	time (s)
SVMLap	228.57	± 1.87	37.69
SVMGauss	232.43	± 2.06	37.56
SVMGaussHP	243.43	± 4.66	64.98
IBk	336.29	± 8.20	175.71
MLP	379.29	± 6.24	26.32
LinearR	445.71	± 7.14	21.90
LRK(poly)	454.57	± 11.10	44.36
LeastMedSq	503.43	± 10.32	241.07
AR	511.57	± 9.00	20.64
REPTree	521.29	± 9.75	20.51
DecTable	548.14	± 8.94	25.51
DecisionStump	555.00	± 7.92	19.65
LogisticBase	558.43	± 9.25	57.96
LRK(linear)	566.14	± 7.61	58.22
LRK(gaus)	570.43	± 6.79	237.51
SimpleLinearR	575.43	± 5.87	19.53
RBFNetwork	583.43	± 4.46	33.49

Many bots (300 points) Baseline Dimension x2 → 16			
Optimizer	score	Std	time (s)
SVMLap	844.29	± 5.67	46.25
SVMGauss	858.71	± 7.54	46.11
SVMGaussHP	892.00	± 13.28	69.31
MLP	1118.71	± 8.89	48.45
LeastMedSq	1122.86	± 7.90	200.55
IBk	1142.00	± 11.82	195.88
LinearR	1145.71	± 10.08	27.19
AR	1218.71	± 12.62	24.77
DecTable	1228.29	± 15.03	32.71
DecisionStump	1281.14	± 13.34	23.89
LRK(poly)	1294.57	± 16.03	122.58
REPTree	1303.29	± 10.48	25.65
SimpleLinearR	1318.14	± 13.82	24.21
LogisticBase	1333.00	± 11.39	66.76
LRK(linear)	1333.43	± 12.08	77.47
LRK(gaus)	1335.71	± 12.17	268.08
RBFNetwork	1345.57	± 11.98	44.10

Many bots (300 points) Baseline Dimension x3 → 24			
Optimizer	score	Std	time (s)
SVMGauss	1578.29	± 9.59	54.96
SVMLap	1586.43	± 7.49	55.17
SVMGaussHP	1697.29	± 20.56	79.53
MLP	1845.86	± 13.57	54.39
LeastMedSq	1878.71	± 12.45	211.79
LinearR	1880.86	± 12.42	31.85
IBk	1950.86	± 17.09	334.90
AR	1966.14	± 13.63	30.42
DecTable	1976.14	± 18.50	38.11
DecisionStump	2037.57	± 15.21	28.24
SimpleLinearR	2064.86	± 17.80	27.82
REPTree	2074.86	± 14.18	29.76
LogisticBase	2092.14	± 15.05	73.60
LRK(poly)	2098.71	± 12.36	165.87
LRK(linear)	2112.43	± 11.96	90.22
LRK(gaus)	2120.00	± 11.14	290.61
RBFNetwork	2128.29	± 13.31	51.15

Many bots (300 points) Baseline Dimension x4 → 32			
Optimizer	score	Std	time (s)
SVMLap	2339.57	± 12.14	66.94
SVMGauss	2344.14	± 9.79	65.80
SVMGaussHP	2443.86	± 20.38	90.55
LeastMedSq	2647.14	± 16.22	224.39
LinearR	2654.86	± 15.54	37.49
MLP	2656.86	± 13.54	58.27
AR	2706.29	± 17.31	38.48
IBk	2713.43	± 17.29	363.77
DecTable	2749.14	± 21.27	43.80
DecisionStump	2801.00	± 21.69	33.95
REPTree	2841.14	± 17.30	35.77
LRK(poly)	2848.43	± 18.08	183.06
LRK(gaus)	2861.00	± 17.59	310.99
SimpleLinearR	2862.00	± 17.79	33.24
LRK(linear)	2867.29	± 18.74	103.55
LogisticBase	2868.00	± 16.73	82.26
RBFNetwork	2901.71	± 16.82	58.45

Table B.10: Results on the ”Many bots” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 8, 16, 24 or 32). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Arm (300 points) Baseline Dimension x1 → 3			
Optimizer	score	Std	time (s)
SVMLap	11.89	± 0.25	45.74
SVMGauss	11.99	± 0.26	45.61
SVMGaussHP	12.22	± 0.20	106.23
LeastMedSq	14.08	± 0.18	442.33
LinearR	14.13	± 0.20	41.15
MLP	14.44	± 0.21	38.41
IBk	14.80	± 0.19	150.19
RBFNetwork	15.23	± 0.18	56.56
LRK(gaus)	15.31	± 0.18	104.71
SimpleLinearR	15.31	± 0.19	37.68
REPTree	15.48	± 0.18	39.07
DecTable	15.49	± 0.19	47.80
LogisticBase	15.54	± 0.18	149.86
DecisionStump	15.55	± 0.18	36.84
AR	15.58	± 0.19	37.22
LRK(poly)	16.74	± 0.17	74.45
LRK(linear)	16.95	± 0.16	57.67

Arm (300 points) Baseline Dimension x2 → 6			
Optimizer	score	Std	time (s)
SVMGauss	11.96	± 0.24	49.50
SVMLap	11.98	± 0.23	49.78
SVMGaussHP	12.08	± 0.20	104.34
LeastMedSq	12.85	± 0.22	429.69
IBk	13.12	± 0.20	263.33
LinearR	13.18	± 0.22	43.89
RBFNetwork	13.34	± 0.23	61.61
SimpleLinearR	13.42	± 0.22	39.45
MLP	13.47	± 0.21	43.10
DecTable	13.55	± 0.21	51.08
LogisticBase	13.56	± 0.22	186.98
DecisionStump	13.62	± 0.23	38.93
REPTree	13.64	± 0.20	41.32
LRK(gaus)	13.70	± 0.21	112.58
AR	13.73	± 0.22	39.75
LRK(poly)	14.48	± 0.20	77.84
LRK(linear)	14.82	± 0.19	63.65

Arm (300 points) Baseline Dimension x3 → 9			
Optimizer	score	Std	time (s)
LeastMedSq	12.40	± 0.20	408.14
IBk	12.49	± 0.20	237.23
LinearR	12.60	± 0.22	45.86
SVMGaussHP	12.62	± 0.19	109.07
SVMGauss	12.66	± 0.21	52.60
DecisionStump	12.67	± 0.21	42.00
SVMLap	12.67	± 0.21	52.81
SimpleLinearR	12.68	± 0.19	42.17
LRK(linear)	12.69	± 0.21	76.49
DecTable	12.71	± 0.18	54.49
LogisticBase	12.71	± 0.20	190.06
LRK(gaus)	12.74	± 0.19	132.16
RBFNetwork	12.77	± 0.20	66.54
AR	12.79	± 0.20	43.68
REPTree	12.80	± 0.20	43.62
LRK(poly)	12.86	± 0.18	112.07
MLP	13.28	± 0.19	48.35

Arm (300 points) Baseline Dimension x4 → 12			
Optimizer	score	Std	time (s)
LeastMedSq	11.87	± 0.18	399.03
SVMGauss	11.95	± 0.19	57.06
SVMLap	11.99	± 0.18	57.51
RBFNetwork	12.11	± 0.20	72.01
LRK(linear)	12.12	± 0.18	87.95
DecisionStump	12.17	± 0.18	45.03
LRK(poly)	12.17	± 0.18	149.59
SVMGaussHP	12.18	± 0.19	114.93
LRK(gaus)	12.20	± 0.18	167.58
LogisticBase	12.21	± 0.18	198.82
AR	12.22	± 0.18	47.17
SimpleLinearR	12.22	± 0.19	44.67
DecTable	12.23	± 0.18	57.93
REPTree	12.24	± 0.18	46.62
IBk	12.27	± 0.18	435.12
LinearR	12.55	± 0.19	48.85
MLP	12.89	± 0.18	52.46

Table B.11: Results on the "Arm" problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 3, 6, 9 or 12). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Away (300 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
IBk	1.80	± 0.23	163.50
SVMLap	1.89	± 0.23	44.83
SVMGauss	1.96	± 0.25	44.58
MLP	3.01	± 0.42	40.05
SVMGaussHP	3.38	± 0.51	127.13
LRK(gaus)	3.71	± 0.63	116.37
AR	5.02	± 0.66	36.93
REPTree	5.49	± 0.64	38.31
DecTable	6.28	± 0.75	50.43
DecisionStump	6.40	± 0.77	35.70
LogisticBase	7.02	± 0.81	133.87
LinearR	7.09	± 0.67	42.14
RBFNetwork	7.10	± 0.78	62.02
SimpleLinearR	7.11	± 0.67	37.16
LRK(poly)	7.54	± 0.56	95.09
LeastMedSq	7.86	± 0.81	507.22
LRK(linear)	9.10	± 0.66	68.23

Away (300 points) Baseline Dimension x2 → 4			
Optimizer	score	Std	time (s)
SVMLap	1.33	± 0.22	59.95
SVMGauss	1.36	± 0.21	59.74
SVMGaussHP	1.44	± 0.23	132.84
MLP	2.10	± 0.34	48.13
IBk	2.46	± 0.39	215.29
LRK(poly)	3.37	± 0.41	110.49
LRK(gaus)	3.44	± 0.59	122.93
AR	4.05	± 0.61	38.11
DecTable	4.39	± 0.59	54.15
SimpleLinearR	4.54	± 0.49	38.72
REPTree	4.59	± 0.63	39.48
DecisionStump	4.68	± 0.64	37.56
LinearR	4.92	± 0.62	43.67
LogisticBase	5.01	± 0.65	126.44
RBFNetwork	5.13	± 0.61	67.88
LeastMedSq	5.93	± 0.62	519.33
LRK(linear)	6.43	± 0.42	75.27

Away (300 points) Baseline Dimension x3 → 6			
Optimizer	score	Std	time (s)
SVMGauss	0.91	± 0.18	70.72
SVMLap	0.93	± 0.18	70.98
SVMGaussHP	1.37	± 0.23	142.87
LRK(poly)	1.64	± 0.23	125.17
MLP	1.68	± 0.32	57.47
IBk	1.81	± 0.26	248.93
AR	2.86	± 0.45	41.36
LRK(gaus)	3.01	± 0.43	142.57
DecisionStump	3.12	± 0.48	39.64
DecTable	3.24	± 0.47	55.52
REPTree	3.28	± 0.47	41.20
SimpleLinearR	3.35	± 0.46	39.87
LogisticBase	3.45	± 0.50	119.45
LeastMedSq	3.86	± 0.51	473.13
RBFNetwork	3.86	± 0.48	69.21
LinearR	3.94	± 0.46	45.33
LRK(linear)	4.71	± 0.44	80.25

Away (300 points) Baseline Dimension x4 → 8			
Optimizer	score	Std	time (s)
SVMLap	0.61	± 0.13	73.93
SVMGauss	0.70	± 0.16	73.44
SVMGaussHP	0.87	± 0.18	135.68
MLP	1.26	± 0.22	68.00
LRK(poly)	1.29	± 0.25	155.97
IBk	1.41	± 0.23	298.60
LogisticBase	2.06	± 0.33	116.55
REPTree	2.10	± 0.37	50.18
RBFNetwork	2.11	± 0.35	73.85
DecTable	2.15	± 0.35	67.76
LRK(gaus)	2.15	± 0.40	178.65
DecisionStump	2.19	± 0.32	42.39
AR	2.20	± 0.35	43.14
SimpleLinearR	2.29	± 0.35	46.48
LinearR	2.59	± 0.33	50.06
LeastMedSq	2.96	± 0.35	488.00
LRK(linear)	3.14	± 0.40	95.82

Table B.12: Results on the ”Away” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 2, 4, 6 or 8). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Stock Management (500 points) Baseline Dimension x1 → 4 8			
Optimizer	score	Std	time (s)
SVMLap	1859.41	± 1.65	66.61
SVMGauss	1859.48	± 1.62	66.16
SVMGaussHP	2111.34	± 25.38	270.42
LinearR	2486.26	± 1.82	60.85
IBk	2517.44	± 4.69	574.92
LRK(linear)	2748.23	± 2.22	177.74
DecTable	2849.66	± 2.87	72.94
LRK(gaus)	2876.35	± 9.87	390.70
AR	2903.80	± 7.26	53.70
REPTree	2997.14	± 2.73	54.90
LRK(poly)	2998.82	± 4.26	338.82
RBFNetwork	3031.20	± 1.72	91.19
LogisticBase	3031.21	± 0	470.23
DecisionStump	3031.27	± 0	52.59
LeastMedSq	3031.27	± 0	483.15
MLP	3031.27	± 0	68.18
RBFNetwork	3031.27	± 0	105.35

Stock Management (500 points) Baseline Dimension x2 → 8 8			
Optimizer	score	Std	time (s)
IBk	2941.56	± 2.89	834.21
SVMGaussHP	2983.52	± 8.12	202.79
SVMLap	2996.27	± 1.54	72.45
SVMGauss	2996.40	± 1.55	71.93
REPTree	3013.33	± 1.81	63.66
AR	3023.55	± 1.87	63.43
LinearR	3026.17	± 1.70	70.78
DecTable	3030.37	± 1.67	78.74
MLP	3031.17	± 0	84.28
DecisionStump	3031.27	± 0	62.80
LeastMedSq	3031.27	± 0	518.47
LogisticBase	3031.27	± 0	368.97
LRK(gaus)	3031.27	± 0	421.91
LRK(linear)	3031.27	± 0	209.79
LRK(poly)	3031.27	± 0	288.54
RBFNetwork	3031.27	± 0	105.35
SimpleLinearR	3031.27	± 0	61.39

Stock Management (500 points) Baseline Dimension x3 → 12			
Optimizer	score	Std	time (s)
IBk	2934.70	± 2.79	1121.49
SVMGaussHP	3002.72	± 4.08	180.71
SVMGauss	3017.09	± 1.67	77.71
SVMLap	3017.24	± 1.67	78.17
LinearR	3025.58	± 1.69	75.95
AR	3030.35	± 1.76	70.49
DecTable	3030.84	± 0	89.03
DecisionStump	3031.27	± 0	69.12
LeastMedSq	3031.27	± 0	498.46
LogisticBase	3031.27	± 0	343.47
LRK(gaus)	3031.27	± 0	466.56
LRK(linear)	3031.27	± 0	240.74
LRK(poly)	3031.27	± 0	310.58
MLP	3031.27	± 0	78.99
RBFNetwork	3031.27	± 0	118.12
REPTree	3031.27	± 0	70.57
SimpleLinearR	3031.27	± 0	66.21

Stock Management (500 points) Baseline Dimension x4 → 16			
Optimizer	score	Std	time (s)
AR	3031.27	± 0	97.02
DecisionStump	3031.27	± 0	91.00
DecTable	3031.27	± 0	99.86
IBk	3031.27	± 0	1516.11
LeastMedSq	3031.27	± 0	431.71
LinearR	3031.27	± 0	97.64
LogisticBase	3031.27	± 0	285.45
LRK(gaus)	3031.27	± 0	495.83
LRK(linear)	3031.27	± 0	252.15
LRK(poly)	3031.27	± 0	308.57
MLP	3031.27	± 0	80.65
RBFNetwork	3031.27	± 0	128.35
REPTree	3031.27	± 0	91.54
SimpleLinearR	3031.27	± 0	82.11
SVMLap	3031.27	± 0	62.83
SVMGauss	3031.27	± 0	60.24
SVMGaussHP	3031.27	± 0	78.95

Table B.13: Results on the ”Stock Management” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 4, 8, 12 or 16). The absolute value of the score (cost) can’t be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Stock Management V2 (500 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
SVMGaussHP	339.86	± 3.31	212.83
SVMLap	361.14	± 4.95	31.63
SVMGauss	374.69	± 6.14	31.47
IBk	479.92	± 6.33	233.34
LinearR	691.03	± 3.94	35.79
DecTable	864.53	± 19.24	39.35
REPTree	921.26	± 13.85	33.72
LRK(linear)	1182.30	± 11.48	95.04
SimpleLinearR	1226.95	± 2.17	31.94
AR	1250.32	± 11.96	31.71
LRK(gaus)	1369.07	± 26.72	185.17
DecisionStump	1599.44	± 4.78	32.00
LRK(poly)	1602.45	± 23.28	177.96
LeastMedSq	1650.96	± 2.16	276.08
LogisticBase	1655.84	± 1.49	195.23
MLP	1657.41	± 1.37	31.75
RBFNetwork	1667.60	± 2.29	49.21

Stock Management V2 (500 points) Baseline Dimension x2 → 4			
Optimizer	score	Std	time (s)
SimpleLinearR	247.62	± 4.10	34.12
SVMGauss	263.73	± 10.04	37.14
LeastMedSq	267.68	± 5.48	225.96
SVMLap	271.09	± 9.21	37.60
LinearR	275.82	± 5.73	37.82
SVMGaussHP	277.99	± 8.47	141.80
LRK(linear)	293.67	± 4.41	107.97
IBk	299.37	± 5.10	452.52
MLP	301.89	± 3.07	33.00
REPTree	302.46	± 3.26	36.62
LogisticBase	303.37	± 3.46	244.50
LRK(gaus)	303.42	± 3.53	233.82
AR	307.50	± 2.94	33.44
DecisionStump	308.56	± 2.97	33.80
DecTable	311.65	± 3.38	43.82
RBFNetwork	312.77	± 4.29	57.79
LRK(poly)	314.62	± 3.67	174.18

Table B.14: Results on the "Stock Management V2" problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$ and $\times 2$ which give dimension 2 and 4). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Fast Obstacle Avoidance (500 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
IBk	550.00	± 0.00	130.73
SVMGauss	572.86	± 4.74	34.88
SVMLap	574.29	± 7.25	35.02
REPTree	695.00	± 10.92	27.10
DecTable	700.00	± 9.65	34.22
SVMGaussHP	866.43	± 21.96	117.75
MLP	882.14	± 23.38	28.06
LeastMedSq	995.00	± 5.00	306.36
LRK(linear)	995.00	± 5.00	123.92
AR	996.43	± 3.57	26.29
DecisionStump	1000.00	± 0.00	25.79
LinearR	1000.00	± 0.00	28.67
LogisticBase	1000.00	± 0.00	76.49
RBFNetwork	1000.00	± 0.00	41.90
SimpleLinearR	1000.00	± 0.00	26.21
LRK(gaus)	1000.71	± 0.71	150.70
LRK(poly)	1015.71	± 4.26	90.88

Many-obstacle avoidance (500 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
LRK(poly)	60.43	± 0.20	91.89
MLP	60.86	± 0.23	28.09
IBk	63.43	± 0.33	131.40
SVMGaussHP	75.86	± 2.19	296.54
SVMGauss	76.14	± 1.77	31.59
SVMLap	78.50	± 1.83	31.73
REPTree	87.93	± 1.24	27.91
AR	91.64	± 0.95	27.60
LeastMedSq	99.00	± 0.71	321.15
DecTable	99.50	± 0.25	35.43
LRK(linear)	99.50	± 0.31	126.62
DecisionStump	99.79	± 0.16	26.90
LinearR	100.00	± 0.00	29.94
LogisticBase	100.00	± 0.00	65.34
LRK(gaus)	100.00	± 0.00	152.01
RBFNetwork	100.00	± 0.00	43.41
SimpleLinearR	100.00	± 0.00	27.21

Table B.15: Results on the ”Fast obstacle avoidance” (left) and ”Many-obstacle avoidance” (right) problems (see section 3.2.2 for details). These are in dimension 2. See table 3.3 for a summary.

Many bots (500 points) Baseline Dimension x1 → 8			
Optimizer	score	Std	time (s)
SVMGauss	214.14	± 2.40	80.24
SVMLap	218.86	± 2.18	80.54
SVMGaussHP	225.57	± 3.38	143.40
IBk	332.00	± 8.50	306.97
MLP	345.71	± 7.87	39.05
LRK(poly)	418.43	± 9.06	112.23
LinearR	442.43	± 5.97	37.28
AR	481.71	± 10.72	34.99
LeastMedSq	500.57	± 10.62	299.67
REPTree	505.57	± 10.30	34.09
DecTable	523.57	± 9.78	44.43
DecisionStump	544.14	± 8.38	33.39
SimpleLinearR	564.57	± 8.36	32.92
RBFNetwork	564.86	± 7.88	55.20
LRK(linear)	565.57	± 6.81	153.11
LogisticBase	573.00	± 6.22	85.80
LRK(gaus)	574.00	± 7.06	660.41

Many bots (500 points) Baseline Dimension x2 → 16			
Optimizer	score	Std	time (s)
SVMLap	804.00	± 6.16	105.19
SVMGauss	816.43	± 6.40	104.73
SVMGaussHP	851.43	± 11.08	158.67
MLP	1081.71	± 9.30	62.80
IBk	1117.71	± 13.08	511.99
LeastMedSq	1134.29	± 8.36	256.42
LinearR	1139.14	± 8.81	46.51
AR	1196.86	± 13.79	42.71
DecTable	1210.57	± 14.00	53.73
DecisionStump	1263.71	± 13.50	41.11
REPTree	1263.71	± 13.39	42.87
LRK(poly)	1294.71	± 13.22	290.92
SimpleLinearR	1295.29	± 14.91	40.89
LogisticBase	1329.57	± 10.99	100.83
LRK(gaus)	1338.14	± 11.18	734.23
LRK(linear)	1338.43	± 13.53	194.69
RBFNetwork	1339.57	± 10.86	69.96

Many bots (500 points) Baseline Dimension x3 → 24			
Optimizer	score	Std	time (s)
SVMGauss	1517.57	± 8.76	125.01
SVMLap	1532.43	± 8.66	125.12
SVMGaussHP	1593.86	± 13.37	173.61
MLP	1838.57	± 13.63	86.01
LeastMedSq	1840.14	± 13.88	273.57
LinearR	1877.00	± 12.99	51.90
AR	1898.00	± 14.51	51.47
IBk	1906.86	± 18.62	771.83
DecTable	1912.57	± 17.29	62.01
DecisionStump	2019.43	± 15.58	47.62
REPTree	2056.00	± 14.81	50.34
SimpleLinearR	2059.00	± 17.09	46.63
LRK(gaus)	2067.71	± 17.31	789.34
LRK(poly)	2070.86	± 15.53	424.52
LRK(linear)	2079.43	± 15.39	229.45
LogisticBase	2111.86	± 13.32	102.29
RBFNetwork	2114.14	± 13.90	77.49

Many bots (500 points) Baseline Dimension x4 → 32			
Optimizer	score	Std	time (s)
SVMLap	2281.71	± 10.79	151.32
SVMGauss	2296.71	± 9.78	150.00
SVMGaussHP	2343.29	± 16.16	198.40
MLP	2601.57	± 15.74	102.63
LinearR	2626.86	± 16.59	64.58
LeastMedSq	2633.86	± 13.94	321.21
AR	2654.29	± 17.13	70.37
DecTable	2666.43	± 21.25	74.47
IBk	2671.43	± 20.57	1009.40
DecisionStump	2765.43	± 20.07	67.60
LRK(poly)	2835.71	± 19.66	491.09
LRK(linear)	2846.29	± 18.84	266.70
SimpleLinearR	2848.43	± 18.56	57.00
REPTree	2858.14	± 17.33	61.75
LRK(gaus)	2861.71	± 19.12	847.33
LogisticBase	2864.14	± 15.96	123.93
RBFNetwork	2909.29	± 15.03	94.80

Table B.16: Results on the ”Many bots” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 8, 16, 24 or 32). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Arm (500 points) Baseline Dimension x1 → 3			
Optimizer	score	Std	time (s)
SVMLap	11.34	± 0.23	93.16
SVMGauss	11.42	± 0.23	92.66
SVMGaussHP	11.61	± 0.17	253.90
MLP	13.22	± 0.19	63.08
LinearR	13.85	± 0.20	68.39
LeastMedSq	14.07	± 0.19	481.90
IBk	14.53	± 0.19	372.62
SimpleLinearR	15.06	± 0.20	62.56
LRK(gaus)	15.12	± 0.18	253.73
DecTable	15.48	± 0.18	79.04
LogisticBase	15.51	± 0.19	276.46
REPTree	15.51	± 0.18	63.97
RBFNetwork	15.53	± 0.18	92.71
DecisionStump	15.56	± 0.18	61.71
AR	15.58	± 0.19	62.25
LRK(linear)	17.28	± 0.13	128.44
LRK(poly)	17.30	± 0.15	175.78

Arm (500 points) Baseline Dimension x2 → 6			
Optimizer	score	Std	time (s)
SVMGauss	10.77	± 0.24	101.44
SVMLap	10.83	± 0.24	102.11
SVMGaussHP	11.03	± 0.19	220.95
LeastMedSq	12.41	± 0.23	544.58
LinearR	12.69	± 0.21	73.07
IBk	13.19	± 0.21	700.59
MLP	13.31	± 0.22	70.04
SimpleLinearR	13.46	± 0.21	66.53
LRK(gaus)	13.48	± 0.21	271.38
RBFNetwork	13.59	± 0.22	103.78
LogisticBase	13.64	± 0.21	316.58
DecTable	13.67	± 0.20	82.54
AR	13.69	± 0.22	66.04
DecisionStump	13.71	± 0.21	65.35
REPTree	13.71	± 0.21	68.83
LRK(poly)	15.03	± 0.20	188.00
LRK(linear)	15.07	± 0.18	144.00

Arm (500 points) Baseline Dimension x3 → 9			
Optimizer	score	Std	time (s)
SVMGaussHP	12.01	± 0.19	232.92
SVMGauss	12.06	± 0.21	106.93
LeastMedSq	12.07	± 0.21	541.22
SVMLap	12.08	± 0.21	107.79
IBk	12.55	± 0.19	604.22
SimpleLinearR	12.71	± 0.20	71.48
DecisionStump	12.73	± 0.19	70.40
RBFNetwork	12.76	± 0.20	109.35
LogisticBase	12.77	± 0.19	298.10
LRK(gaus)	12.77	± 0.20	314.76
AR	12.81	± 0.20	71.50
DecTable	12.83	± 0.20	87.72
REPTree	12.84	± 0.20	74.41
LRK(poly)	12.92	± 0.18	227.75
MLP	13.22	± 0.20	76.70
LRK(linear)	13.38	± 0.19	170.07
LinearR	13.94	± 0.22	78.65

Arm (500 points) Baseline Dimension x4 → 12			
Optimizer	score	Std	time (s)
LeastMedSq	11.69	± 0.20	526.61
SVMLap	11.84	± 0.18	116.62
SVMGaussHP	11.86	± 0.18	244.94
SVMGauss	11.89	± 0.19	116.15
LRK(poly)	12.03	± 0.18	256.57
SimpleLinearR	12.15	± 0.18	75.76
IBk	12.18	± 0.18	772.09
LogisticBase	12.18	± 0.19	315.02
REPTree	12.19	± 0.19	79.05
DecisionStump	12.22	± 0.19	76.82
LRK(gaus)	12.22	± 0.20	349.85
RBFNetwork	12.23	± 0.19	114.39
DecTable	12.28	± 0.19	92.61
AR	12.30	± 0.19	80.19
LRK(linear)	12.55	± 0.18	185.90
MLP	13.04	± 0.20	81.64
LinearR	14.07	± 0.22	84.40

Table B.17: Results on the "Arm" problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 3, 6, 9 or 12). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Away (500 points) Baseline Dimension x1 → 2			
Optimizer	score	Std	time (s)
IBk	1.67	± 0.24	398.68
SVMLap	1.68	± 0.20	84.23
SVMGauss	1.75	± 0.21	83.70
MLP	2.95	± 0.41	63.92
SVMGaussHP	3.44	± 0.52	275.97
LRK(gaus)	3.74	± 0.63	302.21
AR	4.81	± 0.59	59.82
REPTree	5.51	± 0.61	61.52
DecTable	5.69	± 0.67	82.64
SimpleLinearR	6.46	± 0.60	60.46
DecisionStump	6.54	± 0.78	59.05
LRK(poly)	6.66	± 0.63	227.38
LinearR	6.74	± 0.65	68.45
LogisticBase	6.80	± 0.81	212.31
RBFNetwork	7.27	± 0.75	106.86
LeastMedSq	7.49	± 0.78	579.93
LRK(linear)	9.15	± 0.71	167.87

Away (500 points) Baseline Dimension x2 → 4			
Optimizer	score	Std	time (s)
SVMGauss	1.27	± 0.21	126.13
SVMLap	1.29	± 0.23	126.87
SVMGaussHP	1.37	± 0.20	304.96
MLP	1.99	± 0.33	75.24
IBk	2.19	± 0.37	521.14
LRK(gaus)	2.50	± 0.50	311.53
LRK(poly)	2.83	± 0.37	250.74
AR	3.97	± 0.58	63.75
DecTable	4.51	± 0.64	83.51
DecisionStump	4.65	± 0.65	62.76
REPTree	4.66	± 0.61	65.57
SimpleLinearR	4.66	± 0.57	63.96
LogisticBase	4.86	± 0.63	173.21
RBFNetwork	5.41	± 0.68	112.86
LinearR	5.60	± 0.65	72.38
LeastMedSq	6.09	± 0.65	652.79
LRK(linear)	6.28	± 0.43	183.81

Away (500 points) Baseline Dimension x3 → 6			
Optimizer	score	Std	time (s)
SVMLap	0.76	± 0.14	155.13
SVMGauss	0.93	± 0.17	154.46
SVMGaussHP	1.03	± 0.17	349.98
MLP	1.19	± 0.21	89.47
LRK(poly)	1.22	± 0.19	283.96
IBk	1.86	± 0.35	634.78
LRK(gaus)	2.81	± 0.43	344.02
DecTable	3.02	± 0.44	85.10
AR	3.09	± 0.45	70.37
DecisionStump	3.45	± 0.49	66.93
REPTree	3.46	± 0.52	68.28
SimpleLinearR	3.54	± 0.46	65.79
RBFNetwork	3.72	± 0.53	120.06
LogisticBase	3.76	± 0.50	163.01
LinearR	4.19	± 0.51	76.61
LeastMedSq	4.58	± 0.47	613.44
LRK(linear)	4.64	± 0.46	201.67

Away (500 points) Baseline Dimension x4 → 8			
Optimizer	score	Std	time (s)
SVMGauss	0.46	± 0.11	162.42
SVMLap	0.47	± 0.10	163.29
SVMGaussHP	0.68	± 0.16	310.56
LRK(poly)	0.86	± 0.15	373.91
MLP	1.01	± 0.17	98.91
IBk	1.47	± 0.23	727.20
LRK(gaus)	1.84	± 0.27	462.50
DecisionStump	2.01	± 0.32	71.81
AR	2.04	± 0.34	72.47
DecTable	2.06	± 0.35	106.99
SimpleLinearR	2.12	± 0.33	79.51
LogisticBase	2.23	± 0.35	162.26
RBFNetwork	2.26	± 0.35	124.88
LinearR	2.41	± 0.26	81.26
REPTree	2.58	± 0.43	78.50
LeastMedSq	2.93	± 0.40	642.78
LRK(linear)	3.16	± 0.40	241.78

Table B.18: Results on the ”Away” problem (see section 3.2.2 for details). Each table is with a different dimension ($\times 1$, $\times 2$, $\times 3$ and $\times 4$ which give dimension 2, 4, 6 or 8). The absolute value of the score (cost) can't be compared between the different dimensions as increasing the dimension can make the optimal solution more or less costly. Only relative values are relevant. See table 3.3 for a summary.

Appendix C

Summary of results of our Go program

Table C.1 summaries the online results of our program MoGo against other programs or humans at the time of writing (March 2007, updated September 2007)

Event	**Result**
July 2006, Birth of MoGo, CGOS (**9x9**)	**1650** ELO
July 2006 KGS (**9x9**)	**3th/4**
August 2006, KGS (**9x9**)	**4th/10**
August 2006, KGS (**13x13**)	**4th/5**
Mid of August 2006, CGOS (**9x9**)	**1950** ELO, best rank
End of August 2006, CGOS (**9x9**)	**2050** ELO
September 2006, KGS (**19x19**) Formal	**7th/7**
September 2006, KGS (**19x19**) Open	**3th/6**
October 2006, KGS (**9x9**)	**1st/8**
October 2006, KGS (**13x13**)	**1st/7**
November 2006, KGS (**9x9**)	**1st/10**
November 2006, KGS (**13x13**)	**1st/5**
December 2006, KGS (**19x19**) Formal	**2nd/6**
December 2006, KGS (**19x19**) Open	technical problem
December 2006, KGS slow (**19x19**)	**1st/8**
December 2006, CGOS (**9x9**)	**2323** ELO
January 2007, KGS (**9x9**) Against humans	**3/6** wins (*komi* 0.5) against the 45th european player (6 dan)
January 2007, KGS (**9x9**)	**1st/7**
January 2007, KGS (**13x13**)	**1st/10**
February 2007, KGS (**9x9**)	**1st/8**
February 2007, KGS (**13x13**)	**1st/5**
February 2007, KGS (**19x19**) Against humans	**4 kyu**
March 2007, KGS (**19x19**) Formal	**1st/8**
March 2007, KGS (**19x19**) Open	**1st/12**
March 2007, CGOS (**9x9**)	**2466** ELO
April 2007, KGS (**9x9**) Formal	**1st/7**
April 2007, KGS (**13x13**) Open	**2nd/10**
May 2007, KGS (**13x13**) Formal	**3rd/7**
May 2007, KGS (**9x9**) Open	**1st/10**
June 2007, KGS (**19x19**) Formal	**3rd/6**
June 2007, KGS (**19x19**) Open	**1st/6**
July 2007, Olympiads (**19x19**)	**1st/8**
July 2007, Olympiads (**9x9**)	**2nd/10**
July 2007, Human (**9x9**)	**1st** victory against a **professional** human player (5 dan pro).
August 2007, KGS (**19x19**) Against humans	**2 kyu**
August 2007, CGOS (**19x19**)	**2370** ELO

Table C.1: Result of MoGo in different international competitions, against computers and humans. This table contains all the KGS tournaments since the beginning of MoGo and some events.

Bibliography

[AB99] M. Antony and P.L. Bartlett. *Neural network learning : Theoretical Foundations*. Cambridge University Press, 1999.

[ACBF02] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[ACBFS95] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[AJT05] A. Auger, M. Jebalia, and O. Teytaud. Xse: quasi-random mutations for evolution strategies. In *Evolutionary Algorithms*, 2005.

[Aka70] H. Akaike. Statistical predictor identification. *Ann. Inst. Statist. Math.*, 22:203–217, 1970.

[Alb71] J.S. Albus. A theory of cerebellar function. *Mathematical Biosciences*, 10:25–61, 1971.

[AM97] S.M. Aji and R. J. McEliece. A general algorithm for distributing algorithm on a graph. In *Proceedings IEEE International Symposium on Information Theory*, page 6. Ulm, Germany, 1997.

[And93] C.W. Anderson. Q-learning with hidden-unit restarting. In *Advances in Neural Information Processing Systems 5*, pages 81–88, 1993.

[AO06] Peter Auer and Ronald Ortner. Logarithmic online regret bounds for undiscounted reinforcement learning. In *Proceedings of NIPS 2006*, 2006.

[Auz04] Janis Auzins. Direct optimization of experimental designs. In *AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference.*, 2004.

[AVAS04] Naoki Abe, Naval K. Verma, Chidanand Apté, and Robert Schroko. Cross channel optimized marketing by reinforcement learning. In *KDD, Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 767–772, 2004.

[AWM93] C. Atkeson A. W. Moore. Prioritized sweeping : Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[BBS93] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. Technical Report UM-CS-1993-002, , 1993.

[BC01] Bruno Bouzy and Tristan Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[BC05] Bruno Bouzy and Guillaume Chaslot. Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. *In G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, pages 176–181, 2005.

[BC06] Bruno Bouzy and Guillaume Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. In G. Kendall and S. Louis, editors, *IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA*, pages 187–194., 2006.

[BD59] R. Bellman and S. Dreyfus. Functional approximation and dynamic programming. *Math. Tables and other Aids Comp.*, 13:247–251, 1959.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

[Ber95] D.P. Bertsekas. *Dynamic Programming and Optimal Control, vols I and II*. Athena Scientific, 1995.

[Bes] Besag. Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society*, 36:192–236.

[Bey01] H.-G. Beyer. *The Theory of Evolutions Strategies*. Springer, Heidelberg, 2001.

[BH99] C. Boutilier and S. Hanks. Decision-theoretic planning : Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[BH01] Andrew Brown and Geoffrey Hinton. Products of hidden markov models. *Proceedings of Artificial Intelligence and Statistics 2001, Morgan Kaufmann, pp3-11*, 2001.

[BH03] B. Bouzy and B. Helmstetter. Monte carlo go developments, 2003.

[BHS91] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In R. K. Belew and L. B. Booker, editors, *Proceedings of the* 4^{th} *International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1991.

[BK02] C. Borglet and K. Kruse. *Graphical Models - Methods for data analysis and Mining*. John Wiley and Sons, Chichester, UK, 2002.

[BLN95] R. H. Byrd, P. Lu, and J. Noceda. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, 16(5):1190–1208, 1995.

[BLNZ95] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, 1995.

[BMJ+06] Michael Bowling, Peter McCracken, Michael James, James Neufeld, and Dana Wilkinson. Learning predictive state representations using non-blind policies. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 129–136, New York, NY, USA, 2006. ACM Press.

[BNS94] R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representation of quasi-newton matrices and their use in limited memory methods. *Mathematical Programming*, 63(4):129–156, 1994.

[Boo90] M. Boon. A pattern matcher for goliath. *Computer Go*, 13:13–23, 1990.

[BOS04] H.-G. Beyer, M. Olhofer, and B. Sendhoff. On the impact of systematic noise on the evolutionary optimization performance - a sphere model analysis, genetic programming and evolvable machines, vol. 5, no. 4, pp. 327 360. 2004.

[Bou95a] Bruno Bouzy. Les ensembles flous au jeu de go (in french). In *Actes des Rencontres Françaises sur la Logique Floue et ses Applications LFA-95*, pages 334–340, 1995.

[Bou95b] Bruno Bouzy. *Modelisation cognitive du joueur de Go , (in French)*. PhD thesis, University Paris 6, 1995.

[Bou05] Bruno Bouzy. Associating domain-dependent knowledge and monte carlo approaches within a go program. *Information Sciences, Heuristic Search and Computer Game Playing IV, Edited by K. Chen*, (4):247–257, 2005.

[Bro70] C. G. Broyden. The convergence of a class of double-rank minimization algorithms, II: The new algorithm. *J. Inst. Maths. Applics.*, 6:222–231, 1970.

[BRS93] T. Bäck, G. Rudolph, and H.-P. Schwefel. Evolutionary programming and evolution strategies: Similarities and differences. In D. B. Fogel and W. Atmar, editors, *Proceedings of the 2nd Annual Conference on Evolutionary Programming*, pages 11–22. Evolutionary Programming Society, 1993.

[Bru93a] B. Bruegmann. Monte carlo go. 1993.

[Bru93b] B. Bruegmann. Monte-Carlo Go. 1993.

[BS95] T. Bäck and M. Schütz. Evolution strategies for mixed-integer optimization of optical multilayer systems. In J. R. McDonnell, R. G. Reynolds, and D. B. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*. MIT Press, March 1995.

[BT96] D.P. Bertsekas and J.N. Tsitsiklis. Neuro-dynamic programming, athena scientific. 1996.

[BT01] R. I. Brafman and M. Tennenholtz. R-max : a general polynomial time algorithm for near-optimal reinforcement learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 953–958, 2001.

[BTW98] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal*, 21(2):84–99, 1998.

[Bun91] W. L. Buntine. Theory refinement of Bayesian networks. In *Uncertainty in Artificial Intelligence*, pages 52–60, 1991.

[Bur99] M. Buro. From simple features to sophisticated evaluation functions. In *1st International Conference on Computers and Games*, pages 126–145, 1999.

[Bäc95] T. Bäck. *Evolutionary Algorithms in theory and practice*. New-York:Oxford University Press, 1995.

[Cai07] Chen Cai. An approximate dynamic programming strategy for responsive traffic signal control. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)*, pages 303–310, 2007.

[Caz00] Tristan Cazenave. Abstract proof search. *Computers and Games, Hamamatsu, 2000*, pages 39–54, 2000.

[CB01] R. Collobert and S. Bengio. Svmtorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research*, 1:143–160, 2001.

[CBG95] R. Dearden C. Boutilier and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95*, pages 1104–1111, 1995.

[CBL97a] Jie Cheng, David Bell, and Weiru Liu. An algorithm for bayesian network construction from data. In *6th International Workshop on Artificial Intelligence and Statistics*, pages 83–90, 1997.

[CBL97b] Jie Cheng, David Bell, and Weiru Liu. Learning belief networks from data : An information theory based approach. In *sixth ACM International Conference on Information and Knowledge Management CIKM*, pages 325–331, 1997.

[CD06] Laetitia Chapel and Guillaume Deffuant. Svm viability controller active learning. In *Kernel machines for reinforcement learning workshop, Pittsburgh, PA*, 2006.

[CGJ95a] D. A. Cohn, Z. Ghahramani, and M. I. Jordan. Active learning with statistical models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 705–712. The MIT Press, 1995.

[CGJ95b] David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan. Active learning with statistical models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 705–712. The MIT Press, 1995.

[CGK+02] Jie Cheng, Russell Greiner, Jonathan Kelly, David Bell, and Weiru Liu. Learning bayesian networks from data : An information theory based approach. *Artificial Intelligence*, 137:43–90, 2002.

[CH92a] G. Cooper and E. Hersovits. A bayesian method for the induction of probabilistic networks from data. In *Machine Learning*, volume 9, pages 309–347, 1992.

[CH92b] Gregory F. Cooper and Edward Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, 1992.

[CH97] David Maxwell Chickering and David Heckerman. Efficient approximations for the marginal likelihood of bayesian networks with hidden variables. *Mach. Learn.*, 29:181–212, 1997.

[CH05] Tristan Cazenave and Bernard Helmstetter. Combining tactical search and monte-carlo in the game of go. *IEEE CIG 2005*, pages 171–175, 2005.

[Chi95] David Chickering. A transformational characterization of equivalent bayesian network structures. In *Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 87–98, San Francisco, CA, 1995. Morgan Kaufmann.

[Chi96] D. M. Chickering. Learning bayesian networks is np-complete. *AI & STAT*, 1996.

[Chi02a] David Maxwell Chickering. Learning equivalence classes of bayesian-network structures. *J. Mach. Learn. Res.*, 2:445–498, 2002.

[Chi02b] David Maxwell Chickering. Optimal structure identification with greedy search. *Journal of Machine Learning Research*, 3:507–554, 2002.

[CM02] David Maxwell Chickering and Christopher Meek. Finding optimal bayesian networks. In *UAI*, pages 94–102, 2002.

[CM04] C. Cervellera and M. Muselli. A deterministic learning approach based on discrepancy. In *IEEE transactions on Neural Networks 15*, pages 533–544, 2004.

[Con76] J. H. Conway. *On Numbers And Games*. Academic Press, 1976.

[Cou] R. Coulom. High-accuracy value-function approximation with neural networks. In *Esann 2004*.

[Cou02] Rémi Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In Paolo Ciancarini and H. Jaap van den Herik, editors, *5th International Conference on Computer and Games, 2006-05-29*, Turin, Italy, 2006.

[Cou07] Rémi Coulom. Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*, pages 113–124, 2007.

[Coz00] F.G. Cozman. Generalizing variable elimination in bayesian networks. In *Proceedings of the Iberamia/Sbia 2000 Workshops (Workshop on Probabilistic Reasoning in Artificial Intelligence*, pages 27–32. Editora Tec Art, Sao Paulo, 2000.

[CSB+06] Guillaume Chaslot, J.-T. Saito, Bruno Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 83–90, 2006.

[CST97] A. Conn, K. Scheinberg, and L. Toint. Recent progress in unconstrained nonlinear optimization without derivatives, 1997.

[CT95] John G. Cleary and Leonard E. Trigg. K*: an instance-based learner using an entropic distance measure. In *Proc. 12th International Conference on Machine Learning*, pages 108–114. Morgan Kaufmann, 1995.

[CVPL05] N. S. Kaisare C. V. Peroni and J. H. Lee. Optimal control of a fed batch bioreactor using simulation-based approximate dynamic programming. *IEEE Transactions on Control Systems Technology*, 13:786–790, Sept 2005.

[CWB+07] Guillaume Chaslot, Mark H.M. Winands, Bruno Bouzy, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik. Progressive Strategies for Monte-Carlo Tree Search. In Ken Chen et al., editors, *JCIS 2007, Salt Lake City, USA.*, 2007.

[Das97] Sanjoy Dasgupta. The sample complexity of learning fixed-structure bayesian networks. *Mach. Learn.*, 29(2-3):165–180, 1997.

[dCFLGP02] L. M. de Campos, J. M. Fernandez-Luna, J. A. Gamez, and J. M. Puerta. Ant colony optimization for learning bayesian networks. *International Journal of Approximate Reasoning*, 31:291–311, 2002.

[DeJ92] K. A. DeJong. Are genetic algorithms function optimizers ? In R. Manner and B. Manderick, editors, *Proceedings of the 2nd Conference on Parallel Problems Solving from Nature*, pages 3–13. North Holland, 1992.

[DGKL94] L. Devroye, L. Gyorfi, A. Krzyzak, and G. Lugosi. On the strong universal consistency of nearest neighbor regression function estimates. *Annals of Statistics*, 22:1371–1385, 1994.

[dH81] L de Haan. Estimation of the minimum of a function using order statistics. *Journal of the American Statistical Association*, 76:467–469, 1981.

[DS97] D.Precup and R.S. Sutton. Exponentiated gradient methods for reinforcement learning. In *Proc. 14th International Conference on Machine Learning*, pages 272–277. Morgan Kaufmann, 1997.

[EB94] D. Wolfe E. Berlekamp. *Mathematical Go Endgames, Nightmares for the Professional Go Player*. Ishi Press International, San Jose, London, Tokyo, 1994.

[EBN06] Mercedes Esteban-Bravoa and Francisco J. Nogalesb. Solving dynamic stochastic economic models by mathematical programming decomposition methods. *Computers and Operations Research. Part Special Issue: Applications of OR in Finance*, 35:226–240, 2006.

[Enz03] M. Enzenberger. Evaluation in Go by a neural network using soft segmentation. In *10th Advances in Computer Games Conference*, pages 97–108, 2003.

[ES03] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. springer, 2003.

[FG88] J. Fitzpatrick and J. Grefenstette. Genetic algorithms in noisy environments, in machine learning: Special issue on genetic algorithms, p. langley, ed. dordrecht: Kluwer academic publishers, vol. 3, pp. 101 120. 1988.

[FGW99] Nir Friedman, Moises Goldszmidt, and Abraham Wyner. Data analysis with bayesian networks: A bootstrap approach. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 196–205, San Francisco, CA, 1999. Morgan Kaufmann.

[FK03] Nir Friedman and Daphne Koller. Being bayesian about network structure. a bayesian approach to structure discovery in bayesian networks. *Machine Learning*, V50(1):95–125, January 2003.

[Fle70] R. Fletcher. A new approach to variable-metric algorithms. *Computer Journal*, 13:317–322, 1970.

[FM02a] David Filliat and Jean-Arcady Meyer. Global localization and topological map-learning for robot navigation. *Proceedings of the Seventh International Conference on simulation of adaptive behavior : From Animals to Animats (SAB-2002), pages 131-140. The MIT Press*, 2002.

[FM02b] David Filliat and Jean-Arcady Meyer. Map-based navigation in mobile robots. ii a review of map-learning and path planning strategies. 2002.

[Gag05] Christian Gagne. Openbeagle 3.1.0-alpha. 2005.

[GGKH01] Thore Graepel, Mike Goutrié, Marco Krüger, and Ralf Herbrich. Learning on graphs in the game of go. *Lecture Notes in Computer Science*, 2130:347–352, 2001.

[GHHS02] H. Guo, E. Horvitz, W.H. Hsu, and E. Santos. A survey of algorithms for real-time bayesian network inference. *Joint Workshop on Real- Time Decision Support and Diagnosis*, 2002.

[GJ97] Zoubin Ghahramani and Michael I. Jordan. Factorial hidden markov models. *Machine Learning, vol. 29. 1996, pages 245-273*, 1997.

[GM98] Dan Geiger and Christopher Meek. Graphical models and exponential families. In *Proceedings of Fourteenth Conference on Uncertainty in Artificial Intelligence,* Madison, WI, pages 156–165. Morgan Kaufmann, August 1998.

[Gnu99] Gnu go home page. *http://www.gnu.org/software/gnugo/devel.html*, 1999.

[Gol70] D. Goldfarb. A family of variable-metric algorithms derived by variational means. *Mathematics of Computation*, 24:23–26, 1970.

[GRT06] S. Gelly, S. Ruette, and O. Teytaud. Comparison-based algorithms: worst-case optimality, optimality w.r.t a bayesian prior, the intraclass-variance minimization in eda, and implementations with billiards. In *PPSN-BTP workshop*, 2006.

[GWMT06] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.

[HB01] Geoffrey Hinton and Andrew Brown. Training many small hidden markov models. *Proceedings of the Workshop on Innovation in Speech Processing*, 2001.

[HGC94] D. Heckerman, D. Geiger, and M. Chickering. Learning bayesian networks : The combination of knowledge and statistical data. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, 1994.

[Hic98] Fred J. Hickernell. A generalized discrepancy and quadrature error bound. *Mathematics of Computation*, 67(221):299–322, 1998.

[HJ61] R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. *Journal of the ACM, Vol. 8, pp. 212-229*, 1961.

[HO96] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaption. In *Proc. of the IEEE Conference on Evolutionary Computation (CEC 1996)*, pages 312–317. IEEE Press, 1996.

[IP03] Masoumeh T. Izadi and Doina Precup. A planning algorithm for predictive state representation. In *IJCAI '03: Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003.

[Jae00] Herbert Jaeger. Observable operator models for discrete stochastic time series. *Neural Computation*, 12(6):1371–1398, 2000.

[JB05] Y. Jin and J. Branke. Evolutionary optimization in uncertain environments. a survey, ieee transactions on evolutionary computation, vol. 9, no. 3, pp. 303 317. 2005.

[JN06] Martin Janzura and Jan Nielsen. A simulated annealing-based method for learning bayesian networks from statistical data. *Int. J. Intell. Syst.*, 21(3):335–348, 2006.

[Jr.63] Arthur F. Kaupe Jr. Algorithm 178: Direct search. *Communications of the ACM, Vol 6. p.313*, 1963.

[JS04] Michael R. James and Satinder Singh. Learning and discovery of predictive state representations in dynamical systems with reset. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 53, New York, NY, USA, 2004. ACM Press.

[JVN44] O. Morgenstern J. Von Neumann. Theory of games and economic behavior. *Princeton Univ. Press, Princeton*, 1944.

[KA97] R. Kretchmar and C. Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning, 1997.

[KFL01] F. R. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inform. Theory*, 47(2):498–519, 2001.

[KM04] Akihiro Kishimoto and Martin Müller. A general solution to the graph history interaction problem. *Nineteenth National Conference on Artificial Intelligence (AAAI 2004), San jose, CA*, pages 644–649, 2004.

[KMN99] M.J. Kearns, Y. Mansour, and A.Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *IJCAI*, pages 1324–1231, 1999.

[KMRS01] Maarten Keijzer, J. J. Merelo, G. Romero, and Marc Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *Artificial Evolution*, pages 231–244, 2001.

[KMRS02] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: a general purpose evolutionary computation library. In P. Collet, E. Lutton, M. Schoenauer, C. Fonlupt, and J.-K. Hao, editors, *Artificial Evolution'01*, pages 229–241. Springer Verlag, LNCS 2310, 2002.

[Koh95] R. Kohavi. The power of decision tables. In Nada Lavrac and Stefan Wrobel, editors, *Proceedings of the European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 914, pages 174–189, Berlin, Heidelberg, New York, 1995. Springer Verlag.

[KP00] Daphne Koller and Ronald Parr. Policy iteration for factored mdps. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 326–334, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[KR95] S. Keerthi and B. Ravindran. A tutorial survey of reinforcement learning, 1995.

[KS97] Leila Kallel and Marc Schoenauer. Alternative random initialization in genetic algorithms. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[KS04] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2004.

[KS06] L Kocsis and Cs Szepesvari. Bandit based monte-carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[KSW06] L. Kocsis, Cs. Szepesvári, and J. Willemson. Improved monte-carlo search. working paper, 2006.

[KT61] A.-N. Kolmogorov and V.-M. Tikhomirov. ε-entropy and ε-capacity of sets in functional spaces. *Amer. Math. Soc. Translations*, 17:277–364, 1961.

[LB03] John Langford and Avrim Blum. Microchoice bounds and self bounding learning algorithms. *Mach. Learn.*, 51(2):165–179, 2003.

[LBL04] Steven M. LaValle, Michael S. Branicky, and Stephen R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *I. J. Robotic Res.*, 23(7-8):673–692, 2004.

[Lef60] D. Lefkovitz. A strategic pattern recognition program for the game of go. Technical Report 60-243, University of Pennsylvania, the Moore school of Electrical Engineering, Wright Air Development Division, 1960.

[LG94] D. Lewis and W. Gale. Training text classifiers by uncertainty sampling. In *Proceedings of International ACM Conference on Research and Development in Information Retrieval*, pages 3–12, 1994.

[LL02] P. L'Ecuyer and C. Lemieux. Recent advances in randomized quasi-monte carlo methods, 2002.

[LL03] S. R. Lindemann and S. M. LaValle. Incremental low-discrepancy lattice methods for motion planning. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 2920–2927, 2003.

[LMdC01] J. M. Puerta L. M. de Campos. Stochastic local and distributed search algorithms for learning belief networks. In *Proceedings of the III International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Model*, pages 109–115, 2001.

[LMP01] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*, pages 282–289. Morgan Kaufmann, San Francisco, CA, 2001.

[LPY+96] Pedro Larrañaga, Mikel Poza, Yosu Yurramendi, Roberto H. Murga, and Cindy M. H. Kuijpers. Structure learning of bayesian networks by genetic algorithms: A performance analysis of control parameters. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18(9):912–926, 1996.

[LS88] S. Lauritzen and D. Spiegelhalter. Local computations with probabilities on graphical structures and their applications to expert systems. *J. Royal Statistical Society B*, 50:157–224, 1988.

[LW01] F. Liang and W.H. Wong. Real-parameter evolutionary monte carlo with applications in bayesian mixture models. *J. Amer. Statist. Assoc.*, 96:653–666, 2001.

[MK02] S. Singh M. Kearns. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49:209–232, 2002.

[MLD99] James W. Myers, Kathryn B. Laskey, and Kenneth A. DeJong. Learning bayesian networks from incomplete data using evolutionary algorithms. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 458–465, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[MLL01] S. Singh M. L. Littman, R. S. Sutton. Predictive representations of state. In *Advances In Neural Information Processing Systems 14*, 2001.

[MM99a] R. Munos and A. Moore. Variable resolution discretization in optimal control. Technical report, 1999.

[MM99b] Remi Munos and Andrew W. Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *IJCAI*, pages 1348–1355, 1999.

[MR90] Veall MR. Testing for a global maximum in an econometric context. *Econometrica*, 58:1459—1465, 1990.

[MRJL04] Satinder Singh Michael R. James and Michael L. Littman. Planning with predictive state representations. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*, pages 304–311, 2004.

[MS01] J. Moody and M. Saffell. Learning to trade via direct reinforcement. *IEEE-NN*, 12:875–889, July 2001.

[MTKW02] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam : A factored solution to the simultaneous localization and mapping problem. *Proc. 10 th National Conference on Artificial Intelligence*, pages 593–598, 2002.

[Mül02] M. Müller. Computer Go. *Artificial Intelligence*, 134(1–2):145–179, 2002.

[Mun05] R. Munos. Error bounds for approximate value iteration. In *Proceedings of AAAI 2005*, 2005.

[Mun07] Remi Munos. Performance bounds in lp norm for approximate value iteration. *SIAM Journal on Control and Optimization*, 2007.

[Mur02] Kevin Patrick Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, Berkeley, 2002.

[New96] Monty Newborn. *Computer Chess Comes of Age*. Springer-Verlag, 1996.

[Nie92] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. 1992.

[NKJS04] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In S. Thrun, L. Saul, and Cambridge B. Schölkopf, MIT Press, editors, *Advances in Neural Information Processing Systems 16*, 2004.

[NSSS05] A. Nakamura, M. Schmitt, N. Schmitt, and H. Simon. Inner product spaces for bayesian networks. *Journal of Machine Learning Research*, 6:1383–1403, 2005.

[NWL+04] P. Naim, P.-H. Wuillemin, P. Leray, O. Pourret, and A. Becker. *Réseaux bayésiens*. Eyrolles, 2004.

[OS05] J. Otero and L. Sanchez. Induction of descriptive fuzzy classifiers with the logitboost algorithm. soft computing 2005. 2005.

[Owe03] A.B. Owen. *Quasi-Monte Carlo Sampling, A Chapter on QMC for a SIGGRAPH 2003 course*. 2003.

[PA02] P. Fischer P. Auer, N. Cesa-Bianchi. Finite-time analysis of the multiarmed bandit problem. *Machine Learning Journal*, 47(2-3):235–256, 2002.

[PAAV02] O. Procopiuc, P. Agarwal, L. Arge, and J. Vitter. Bkd-tree: A dynamic scalable kd-tree, 2002.

[Pea91] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of plausible inference*. Morgan Kaufmann, 1991.

[Pea00] J. Pearl. Causality: models, reasonings and inference. *Cambridge University Press*, 2000.

[PLL04] J. M. Peña, J. A. Lozano, and P. Larrañaga. Unsupervised learning of bayesian networks via estimation of distribution algorithms: an application to gene expression data clustering. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 12(SUPPLEMENT):63–82, 2004.

[Pow07] W. B. Powell. *Approximation Dynamic Programming for Operations Research: Solving the curse of dimensionality*. Princeton University, 2007.

[PP03] K. Papadaki and W. Powell. An adaptive dynamic programming algorithm for a stochastic multiproduct batch dispatch problem. In *Naval Research Logistic*, pages 742–769, 2003.

[PT06] Scott Proper and Prasad Tadepalli. Scaling model-based average-reward reinforcement learning for product delivery. In *ECML, 17th European Conference on Machine Learning*, pages 735–742, 2006.

[Rab89] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[RB03] Pedro Larrañaga Rosa Blanco, Iñaki Inza. Learning bayesian networks in the space of structures by estimation of distribution algorithms. *International Journal of Intellient Systems*, 18:205–220, 2003.

[Ris78] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[RL87] P. Rousseeuw and A. Leroy. *Robust Regression and Outlier Detection*. John Wiley and Sons, 1987.

[Rob94] Christian Robert. The bayesian choice : a decision theoretic motivation. In *Springer, New York*, 1994.

[Roy01] B. Van Roy. Neuro-dynamic programming: Overview and recent trends. *Handbook of Markov Decision Processes: Methods and Applications*, pages 431–460, 2001.

[RP04] B. Ratitch and D. Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *ECML 2004: 347-358*, 2004.

[Rus97] J. Rust. Using randomization to break the curse of dimensionality. *Econometrica*, 65(3):487–516, 1997.

[RWB05] Liva Ralaivola, Lin Wu, and Pierre Baldi. Svm and pattern-enriched common fate graphs for the game of go. *ESANN 2005*, pages 485–490, 2005.

[SB98] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction.* MIT Press., Cambridge, MA, 1998.

[SBO04] B. Sendhoff, H.-G. Beyer, and M. Olhofer. The influence of stochastic quality functions on evolutionary search, in recent advances in simulated evolution and learning, ser. advances in natural computation, k. tan, m. lim, x. yao, and l. wang, eds. world scientific, pp 152-172. 2004.

[SC98] Satinder Singh and David Cohn. How to dynamically merge markov decision processes. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[SC00] Greg Schohn and David Cohn. Less is more: Active learning with support vector machines. In Pat Langley, editor, *Proceedings of the 17th International Conference on Machine Learning*, pages 839–846. Morgan Kaufmann, 2000.

[Sch78] G Schwartz. Estimating the dimension of a model. In *The annals of Statistics*, volume 6, pages 461–464, 1978.

[SDG+00] V. Stephan, K. Debes, H.-M. Gross, F. Wintrich, and H. Wintrich. A reinforcement learning based neural multi-agentsystem for control of a combustion process. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, volume 6, pages 217–222, 2000.

[SDS94] N. Schraudolph, P. Dayan, and T. Sejnowski. Temporal difference learning of position evaluation in the game of Go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 817–824, San Francisco, 1994. Morgan Kaufmann.

[Ser82] J. Serra. *Image analysis and mathematical morphology*. Academic Press, London, 1982.

[SGS93] Peter Spirtes, Clark Glymour, and Richard Scheines. Causation, prediction, and search. 1993.

[Sha70] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24:647–656, 1970.

[SHJ01a] J. Schaeffer, M. Hlynka, and V. Jussila. Temporal difference learning applied to a high-performance game-playing program. In *17th International Joint Conference on Artificial Intelligence*, pages 529–534, 2001.

[SHJ01b] Jonathan Schaeffer, Markian Hlynka, and Vili Jussila. Temporal difference learning applied to a high-performance game-playing program. In *IJCAI*, pages 529–534, 2001.

[SHL+05] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y.-P. Chen, A. Auger, and S. Tiwari. Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. Technical report, Nanyang Technological University, Singapore, 2005.

[SJLS00] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence results for single-step on-policyreinforcement-learning algorithms. *Mach. Learn.*, 38(3):287–308, 2000.

[SJR04] Satinder Singh, Michael R. James, and Matthew R. Rudary. Predictive state representations: a new theory for modeling dynamical systems. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 512–519, Arlington, Virginia, United States, 2004. AUAI Press.

[SLJ+03] Satinder Singh, Michael L. Littman, Nicholas K. Jong, David Pardoe, and Peter Stone. Learning predictive state representations. In *Proceedings of the Twentieth International Conference on Machine Learning*, August 2003.

[SOS92] H. S. Seung, Manfred Opper, and Haim Sompolinsky. Query by committee. In *Computational Learning Theory*, pages 287–294, 1992.

[Sri00] A. Srinivasan. Low-discrepancy sets for high-dimensional rectangles: A survey. In *Bulletin of the European Association for Theoretical Computer Science*, volume 70, pages 67–76, 2000.

[SS07] David Silver, Richard S. Sutton, and Martin Müller 0003. Reinforcement learning of local shape in the game of go. In Manuela M. Veloso, editor, *IJCAI*, pages 1053–1058, 2007.

[SSM07] D. Silver, R. Sutton, and M. Müller. Reinforcement learning of local shape in the game of Go. In *20th International Joint Conference on Artificial Intelligence*, pages 1053–1058, 2007.

[Sut88] R. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(9):9–44, 1988.

[Sut90] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[Sut96a] R. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044, 1996.

[Sut96b] R.S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press, 1996.

[Suz96] Joe Suzuki. Learning bayesian belief networks based on the minimum description length principle: An efficient algorithm using the b & b technique. In *International Conference on Machine Learning*, pages 462–470, 1996.

[SW98] I.H. Sloan and H. Woźniakowski. When are quasi-Monte Carlo algorithms efficient for high dimensional integrals? *Journal of Complexity*, 14(1):1–33, 1998.

[TD04] Yaroslav Bulatov Thomas Dietterich, Adam Ashenfelter. Training conditional random fields via gradient tree boosting. *International Conference on Machine learning 2004*, 2004.

[TG07] Olivier Teytaud and Sylvain Gelly. Nonlinear programming in approximate dynamic programming - bang-bang solutions, stock-management and unsmooth penalties. In *ICINCO-ICSO Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics, Intelligent Control Systems and Optimization*, pages 47–54, 2007.

[Thr92] S. B. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Pittsburgh, Pennsylvania, 1992.

[Thr02] S. Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002. to appear.

[Tia00] Jin Tian. A branch-and-bound algorithm for mdl learning bayesian networks. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 580–588, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[TMK04] G. Theocharous, K. Murphy, and L. Kaelbling. Representing hierarchical pomdps as dbns for multi-scale robot localization. *Proc. of the IEEE international Conference on Robotics and Automation (ICRA'04)*, 2004.

[Ton01] Simon Tong. *Active Learning: Theory and Applications*. PhD thesis, University of Stanford, 2001.

[TRM01] G. Theocharous, K. Rohanimanesh, and S. Mahadevan. Learning hierarchical partially observable markov decision processes for robot navigation. *Proceedings of the IEEE Conference on Robotics and Automation (ICRA-2001). IEEE Press*, 2001.

[Tsu99] S. Tsutsui. A comparative study on the effects of adding perturbations to phenotypic parameters in genetic algorithms with a robust solution searching scheme, in proceedings of the 1999 ieee system, man, and cybernetics conference smc 99, vol. 3. ieee, pp. 585 591. 1999.

[Tuf96] B. Tuffin. On the use of low discrepancy sequences in monte carlo methods, 1996.

[Vap95a] V. N. Vapnik. *The Nature of Statistical Learning*. Springer Verlag, 1995.

[Vap95b] V.N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, N.Y., 1995.

[VC71] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Theory of probability and its applications*, volume 16, pages 264–280, 1971.

[vHW07] Hado van Hasselt and Marco A. Wiering. Reinforcement learning in continuous action spaces. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)*, pages 272–279, 2007.

[Vid97a] M. Vidyasagar. A theory of learning and generalization. In *Springer*, 1997.

[Vid97b] M. Vidyasagar. *A Theory of Learning and Generalization, with Applications to Neural Networks and Control Systems*. Springer-Verlag, 1997.

[Wal04] Hanna M. Wallach. Conditional random fields: An introduction. Technical Report MS-CIS-04-21, Department of Computer and Information Science, University of Pennsylvania, 2004.

[WF05] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2005.

[Wil78] B. Wilcox. Computer go. *American Go Journal*, 13, 1978.

[WJB02] P. Wocjan, D. Janzing, and T. Beth. Required sample size for learning sparse bayesian networks with many variables. In *LANL e-print cs.LG/0204052*, 2002.

[Wol94] T. Wolf. The program gotools and its computer-generated tsume-go database. In *Proceedings of the first Game Programming Workshop*, pages 59–76, Japan Hakone, 1994.

[Wol00] Thomas Wolf. Forward pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122:59–76, 2000.

[Wri] Margaret H. Wright. Direct search methods: Once scorned, now respectable. *ATT Bell Laboratories, Murray Hill, New Jersey 07974*.

[WW97] G. Wasilkowski and H. Wozniakowski. The exponent of discrepancy is at most 1.4778. *Math. Comp*, 66:1125–1132, 1997.

[YIS99] J. Yoshimoto, S. Ishii, and M. Sato. Application of reinforcement learning to balancing of acrobot, 1999.

[ZBPN94] C. Zhu, R.H. Byrd, P.Lu, and J. Nocedal. L-bfgs-b: a limited memory fortran code for solving bound constrained optimization problems. 1994.

[Zob60] Alfred Zobrist. A new hashing method with application for game playing. Technical Report 88, University of Pennsylvania, the Moore school of Electrical Engineering, Wright Air Development Division, 1960.

[Zob69] Alfred Zobrist. A model of visual organization for the game of go. In AFIPS Press, editor, *Proceedings of AFIPS Spring Joint Computer Conference*, pages 103–111, Montvale, NJ, 1969.

[Zob70] Alfred Zobrist. *Feature extractions and representation for pattern recognition and the game of Go*. PhD thesis, Graduate School of the University of Wisconsin, 1970.